Advances in Database Technology — EDBT 2015

18th International Conference
on Extending Database Technology
Brussels, Belgium, March 23–27, 2015
Proceedings

Editors

Gustavo Alonso
Floris Geerts
Lucian Popa
Pablo Barceló
Jens Teubner
Martín Ugarte
Jan Van den Bussche
Jan Paredaens

open proceedings

Advances in Database Technology — EDBT 2015
Proceedings of the 18th International Conference
on Extending Database Technology
Brussels, Belgium, March 23–27, 2015

Editors

Gustavo Alonso, ETH Zurich, Switzerland
Floris Geerts, University of Antwerp, Belgium
Lucian Popa, IBM Research, USA
Pablo Barceló, University of Chile, Chile
Jens Teubner, TU Dortmund, Germany
Martín Ugarte, Pontificia Universidad Catolica de Chile, Chile
Jan Van den Bussche, Hasselt University, Belgium
Jan Paredaens, University of Antwerp, Belgium

OpenProceedings.org
University of Konstanz
University Library
78457 Konstanz, Germany

OpenProceedings ISBN: 978-3-89318-067-7 DOI of this front matter: `10.5441/002/edbt.2015.01`

Foreword

The 2015 International Conference on Extending Database Technology (EDBT) took place between the 23rd and the 27th of March in Brussels, Belgium. In its 18th edition, EDBT 2015 continued its long tradition of offering an outstanding research venue for the database community where to present and discuss recent contributions.

This year, there were 184 submissions to the research track, 19 to the industrial track, and 25 to the demo track. While these numbers are somewhat lower than in recent editions, the quality of the submissions was very high, which made the job of the Program Committee quite difficult. At the end, the program committee selected for publication 13 demos, 9 industrial papers, and 47 research papers. Of the latter, 5 of them were Vision Papers, shorter papers proposing radically new ideas, which were presented in their own session at the conference.

I would like to take this opportunity to thank all those that have made the 2015 EDBT edition such a success. First and foremost, all the authors of papers submitted to the conference, thereby providing the basis for a strong program, as well as the program committee members for their effort and dedication to study the submissions in detail and engaging in many interesting discussions about the papers, their contributions, their merits, and how to create the best possible program. Special mention should be made of the small committee in charge of deciding the Test of Time Award, in this occasion covering 4 editions of the conference (from 1988 to 1994): Martin Kersten, Christoph Koch, and Guido Moerkotte. They selected the following paper for the award:

> Geo-Relational Algebra: A Model and Query Language for Geometric Database Systems, Ralf Hartmut Güting, University of Hagen, Germany, from EDBT 1988.

I also would like to thank Lucian Popa and Jens Teubner for the great work they have done in running the industrial and demo tracks. Martín Ugarte has done an excellent job with the proceedings, as have Pablo Barceló with the tutorials and Peter Fischer with the workshops. The Conference Chair—Floris Geerts—and the local organizers have also been instrumental in coordinating all the efforts of what is a very complex and demanding endeavor. As a joint EDBT/ICDT Conference, we have four keynote talks, two of which have been proposed by the EDBT community. Thanks go to Christoph Koch and Wolfgang Lehner for their acceptance of our invitation. Finally, Christine Collet and Norman Paton have been instrumental in coordinating the overall effort with the Executive Board of EDBT.

Gustavo Alonso
EDBT 2015 Program Chair

Program Committee Members

Ashraf Aboulnaga (U Waterloo)
Walid Aref (Purdue)
Roger Barga (Microsoft)
Spyros Blanas (Ohio State U)
Steven Blott (DCU)
Michael Böhlen (U Zurich)
Klemens Böhm (KIT)
Philippe Bonnet (ITU)
Luc Bouganim (INRIA)
Nieves Brisaboa (UDC)
Alejandro Buchmann (TU Darmstadt)
K. Selcuk Candan (ASU)
Michael Carey (UC Irvine)
Stefano Ceri (Politecnico Milano)
Lei Chen (HKUST)
Beng Chin Ooi (NUS)
Graham Cormode (U Warwick)
Isabel Cruz (UI Chicago)
Philippe Cudre-Mauroux (U Freibourg)
Sudipto Das (Microsoft)
Khuzaima Daudjee (U Waterloo)
Jens Dittrich (U Saarland)
Michael Duller (Oracle)
Wenfei Fan (U Edinburgh)
Alan Fekete (U Sydney)
Peter Fischer (Uni Freiburg)
Christoph Freytag (HU Berlin)
Johann Gamper (UNIBZ)
Minos Garofalakis (TU Crete)
Boris Glavic (IIT Illinois)
Jiawei Han (UI Urbana Champaign)
Thomas Heinis (Imperial College)
Melanie Herschel (INRIA)
Stratos Idreos (Harvard)
Arantza Illarramendi (UPV/EHU)
U Kang (KAIST)
Arijit Khan (ETH Zurich)
Nick Koudas (U Toronto)
Georg Lausen (Uni Freiburg)
Wolfgang Lehner (TU Dresden)
Hong-Va Leong (PolyU Hong Kong)
Justin Levandoski (Microsoft)
Roy Levin (IBM)
Ninghui Li (Purdue)
Feifei Li (U Utah)
Eric Lo (PolyU Hong Kong)
Guy Lohman (IBM)
Bertram Ludaescher (UC Davis)
Nikos Mamoulis (HKU Hong Kong)
Stefan Manegold (CWI)
Norman May (SAP)
Sharad Mehrotra (UC Irvine)
Rene Müller (IBM)
Thomas Neumann (TU München)
Esther Pacitti (U Montpellier 2)
Themis Palpanas (Paris Descartes)
Olga Papaemmanouil (Brandeis U)
Marta Patiño (Politécnica Madrid)
Torben B. Pedersen (U Aalborg)
Peter Pietzuch (Imperial College)
Evvagelia Pitoura (U Ioannina)
Calton Pu (Georgia Tech)
Philippe Pucheral (INRIA)
Krithi Ramamrithan (IIT Bombay)
Maya Ramanath (IIT Delhi)
Rodolofo Resende (UFMG)
Tore Risch (Uppsala U)
Kenneth Ross (Columbia)
Pierangela Samarati (U Milan)
Marc H. Scholl (U Konstanz)
Heiko Schuldt (U Basel)
Assaf Schuster (Technion IIT)
Thomas Seidl (RWTH Aachen)
Timos Sellis (RMIT)
Liuba Shrira (Brandeis U)
Jianwen Su (UCSB)
Letizia Tanca (Politecnico Milano)
Ernest Teniente (U Poli de Catalunya)
Peter Triantafillou (U Glasgow)
Anthony K.H. Tung (NUS)
Vasilis Vassalos (AUEB)
Marcos Vaz Salles (U Copenhagen)
Yannis Velegrakis (U Trento)
Gottfried Vossen (U Münster)
Kyu-Young Whang (KAIST)
Jef Wijsen (UMONS)
Yoshitaka Yamamoto (U Yamanashi)
Carlo Zaniolo (UCLA)
Wenjie Zhang (UNSW)
Esteban Zimanyi (ULB)

EDBT 2015 Test of Time Award

In 2014, EDBT began awarding the EDBT test-of-time (ToT) award, with the goal of recognizing one paper, or a small number of papers, presented at EDBT earlier and that have best met the "test of time", i.e. that has had the most impact in terms of research, methodology, conceptual contribution, or transfer to practice over the past decade(s). The EDBT ToT award for 2015 will be presented during the EDBT/ICDT 2015 Joint Conference, March 23-27, in Brussels (Belgium). The EDBT 2015 Test of Time Award committee was formed by Martin Kersten (CWI, The Netherlands), Guido Moerkotte (Uni Mannheim, Germany), Christoph Koch (EPFL, Switzerland), all members of the EDBT 2015 PC and chaired by Gustavo Alonso, the EDBT 2015 PC chair.

The committee was charged with selecting a paper or a small number of papers from the proceedings of the following 4 editions: EDBT'88 - Venice, EDBT'90 - Venice, EDBT'92 - Vienna, EDBT'94 - Cambridge.

After careful consideration, the committee has decided to select the following paper, as the EDBT ToT Award winner for 2015:

Geo-Relational Algebra: A Model and Query Language for Geometric Database Systems
by Prof. Dr. Ralf Hartmut Güting, University of Hagen, Germany
Published in the EDBT 1988 proceedings, 506-527

The paper addresses the user's conceptual model of a database system for geometric data. It proposes to extend relational database management systems by integrating geometry at all levels: At the conceptual level, relational algebra is extended to include geometric data types and operators. At the implementation level, the wealth of algorithms and data structures for geometric problems developed in the past decade in the field of Computational Geometry is exploited. The paper starts from a view of relational algebra as a many-sorted algebra which allows to easily embed geometric data types and operators. A concrete algebra for two-dimensional applications is developed. It can be used as a highly expressive retrieval and data manipulation language for geometric as well as standard data. Also, geo-relational database systems and their implementation strategy are discussed.

The committee members unanimously agreed that this paper clearly stands out in terms of relevance, impact, and influence in databases. Of all the papers considered, this is the one that had had the most and longest lasting impact with results that are still relevant today and whose influence can be traced to many real systems and a significant amount of follow up work.

The paper pioneered an important application area well before it became mainstream and did it in a systematic and clean way that has been very influential in both research and practice. Modern commercial systems all support geographic data types that are nowadays used in a wide range of applications and use cases (maps, locations based services, geographic information systems, mobility, etc.).

The selection committee also appreciated very much the cleanliness, completeness, insights, formalism, and systematic treatment of the problem as well as the approach followed by the author in selecting and solving a research problem.

Table of Contents

Online Data Partitioning in Distributed Database Systems

Kaiji Chen
University of Southern Denmark
chen@imada.sdu.dk

Yongluan Zhou
University of Southern Denmark
zhou@imada.sdu.dk

Yu Cao
EMC Labs China
EMC Corporation
yu.cao@emc.com

ABSTRACT

Most of previous studies on automatic database partitioning focus on deriving a (near-)optimal (re)partition scheme according to a specific pair of database and query workload and oversees the problem about how to efficiently deploy the derived partition scheme into the underlying database system. In fact, (re)partition scheme deployment is often non-trivial and challenging, especially in a distributed OLTP system where the repartitioning is expected to take place online without interrupting and disrupting the processing of normal transactions. In this paper, we propose SOAP, a system framework for scheduling online database repartitioning for OLTP workloads. SOAP aims to minimize the time frame of executing the repartition operations while guaranteeing the correctness and performance of the concurrent processing of normal transactions. SOAP packages the repartition operations into repartition transactions, and then mixes them with the normal transactions for holistic scheduling optimization. SOAP utilizes a cost-based approach to rank the repartition transactions' scheduling priorities, and leverages a feedback model in control theory to determine in which order and at which frequency the repartition transactions should be scheduled for execution. When the system is under heavy workload or resource shortage, SOAP takes a further step by allowing repartition operations to piggyback onto the normal transactions so as to mitigate the resource contention. We have built a prototype on top of PostgreSQL and conducted a comprehensive experimental study on Amazon EC2 to validate SOAP's significant performance advantages.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems - Distributed Databases and Transaction Processing

General Terms

Algorithms, Design, Performance, Experimentation

Keywords

Distributed Database Systems, Online Data Partitioning, Transaction Scheduling

1. INTRODUCTION

The difficulty of scaling front-end applications is well known for DBMSs executing highly concurrent workloads. One approach to this problem employed by many Web-based companies is to partition the data and workload across a large number of commodity, shared-nothing servers using a cost-effective, distributed DBMS. The scalability of online transaction processing (OLTP) applications on these DBMSs depends on the existence of an optimal database design, which defines how an application's data and workload is partitioned across the nodes in a cluster, and how queries and transactions are routed to the nodes. This in turn determines the number of transactions that access data stored on each node and how skewed the load is across the cluster. Optimizing these two factors is critical to scaling complex systems: a growing fraction of distributed transactions and load skew can degrade performance by a factor of over ten. Hence, without a proper design, a DBMS will perform no better than a single-node system due to the overhead caused by blocking, inter-node communication, and load balancing issues.

Automatic database partitioning has been extensively researched in the past. As a consequence, nowadays, most DBMSs offer database partitioning design advisory tools. The idea of these tools analyze the workload at a given time and suggest a (near-)optimal repartition scheme in a cost-based or policy-based manner, with the expectation that the system performance can thereby always maintain a consistently high level. It is then the DBA's responsibility to deploy the derived repartition scheme into the underlying database system, which however often posts great challenges to the DBA. On the one hand, the repartition operations should be executed fast enough so that the new partition scheme can start to take effect as soon as possible. However, granting high execution priorities to the repartitioning operations will inevitably slow down or even stall the normal transaction processing on the database system. On the other hand, the repartitioning procedure should be as transparent to the users as possible. In other words, the normal user transactions' correctness must not be violated and the processing performance should not be significantly influenced. Obviously, even skilled DBAs may not be able to easily figure out the best ways of deploying repartition schemes, especially when the workload changes over time

10.5441/002/edbt.2015.02

and has bursts and peaks. As a result, automatic partition scheme deployment satisfying the above requirements is highly desirable. Surprisingly, few previous studies have been devoted to this important research problem.

In this paper, we focus on the problem about how to optimally execute a database repartition plan consisting of a set of repartition operations in a distributed OLTP system, where the repartitioning is expected to take place online without interrupting and disrupting the normal transactions' processing. We propose SOAP, a system framework for scheduling online database repartitioning for OLTP workloads. SOAP aims to minimize the time frame of executing the repartition operations while guaranteeing the correctness and performance of the concurrent normal transaction processing.

SOAP models and groups the repartition operations into repartition transactions, and then mixes them with the normal transactions for holistic scheduling optimization. There are two basic strategies for SOAP to schedule the repartition transactions, which are similar to the techniques used in state-of-the-art database systems' online repartitioning solutions. The first strategy is to maximize the speed of applying the repartition plan and submit all the repartition transactions to the waiting queue with a priority higher than the normal transactions. The second strategy schedules repartition transactions only when the system is idle. Both basic strategies lack the flexibility to find a good trade-off between the two contradicting objectives: maximizing the speed of executing repartition transactions and minimizing the interferences to the processing of normal transactions. As a result, SOAP interleaves the repartition transactions with normal transactions, and leverages feedback models in control theory to determine in which order and at which frequency the repartition transactions should be scheduled for execution.

In the feedback-based method the repartition transactions have the same priority as the normal transactions, hence they will contend with normal transactions for the locks of database objects and significantly increase the system's workload, especially when the system is under heavy loads or resource shortage. To mitigate this issue, SOAP utilizes a piggyback-based approach, which injects repartition operations into the normal transactions. The overhead of acquiring and releasing locks as well as performing the distributed commit protocols incurred by a repartition transaction can be saved if the normal transaction that it piggybacks on will access the same set of database objects.

While the piggyback-based approach consumes less resources, it fails to take use of the available system resources to speed up the repartitioning process. This may leave some resources unused when the system workload is low and there are few transactions to piggyback on. Therefore, SOAP adopts a hybrid approach that is composed by a piggyback module and the feedback module. When the system workload does not use up all the system's resources, we can make use of the available resources to repartition the data before the actual arrival of transactions that will access them, meanwhile the piggyback-based approach will attempt to let the repartition transactions piggyback on the normal transactions when they arrive.

To summarize, we make the following contributions with this work:

- To the best of our knowledge, we are among the first to specifically study the problem of online deploying database partition schemes into OLTP systems.
- We propose a feedback model that realizes dynamic scheduling of the repartition operations.
- We also propose a piggyback approach to execute selected repartition operations within normal transactions to further mitigate the repartitioning overhead.
- We have built a SOAP prototype on top of PostgreSQL, and conducted a comprehensive experimental study on Amazon EC2 that validates SOAP's significant performance advantages.

The rest of this paper is organized as follows. In Section 2, we describe the generic SOAP system architecture, as well as how SOAP realizes online repartitioning for OLTP workloads. In Section 3, we elaborate SOAP's feedback-based, piggyback-based and hybrid approaches of online scheduling repartition operations. Section 4 presents the experiment set-up and experimental results of a SOAP prototype on an Amazon EC2 cluster. We discuss the related works in Section 5 and then conclude in Section 6.

2. SOAP SYSTEM OVERVIEW

In this section, we describe the generic SOAP system architecture, as well as how SOAP realizes online repartitioning for OLTP workloads.

2.1 SOAP System Architecture

Figure 1 shows a SOAP-enabled distributed database architecture providing OLTP services. The clients submit user transactions through a *transaction manager (TM)*, which can be either centralized or distributed.Each submitted transaction will be given a global unique ID by the TM. TM takes care of the processing life-cycle of transactions and guarantees their ACID properties with certain distributed commit protocols and concurrency control protocols. The *query router* maintains the mappings between data partitions and their resident nodes, based on which it routes the incoming transaction queries to the correct nodes for execution. All the submitted transactions will be associated with a scheduling priority and then put into a *processing queue*, where higher-priority transactions will be executed first, while the FIFO policy will be applied to break the tie. The rules of setting priorities are customizable.

The transaction manager, query router and processing queue are common components in most OLTP systems, while SOAP introduces a new component *repartitioner* to coordinate its online database repartitioning for OLTP workloads. In the following subsection, we describe how the repartitioner works.

2.2 Online Database Repartitioning

In this paper, we consider the scenarios where the type of transactions and frequency of OLTP workloads could change over time, so that periodic database repartitioning is required in order to maintain the system performance.

The repartitioner determines when and how the OLTP database should be repartitioned. Its optimizer component periodically extracts the frequency of transactions and their visiting data partitions from the workload history, and then estimates the system throughput and latency in the near

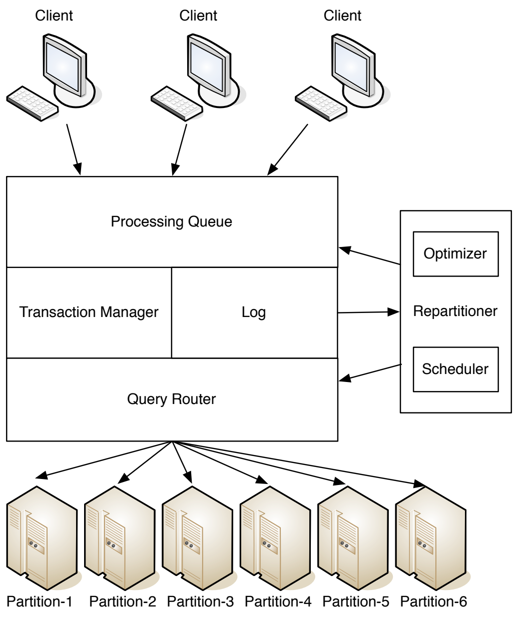


Figure 1: The generic SOAP system architecture

future based on the history. If the estimated system performance is under a predefined threshold, the optimizer will derive a repartition plan in a cost-based manner. The repartition plan could be at the granularity of moving individual tuple or tuples within some ranges or with some hash keys on their attributes. We assume each tuple contains enough information to be positioned by query router. We assume tuple replicas are only made for the purpose of high availability, yet make no assumptions about the replication strategy utilized. Tuple replicas will be distributed over distinct data partitions, and the query router will determine for a transaction which replica of a specific tuple should be visited.

The optimizer will generate three types of repartition operations together with the normal transactions accessing the database objects repartitioned by each of them, i.e. *new replica creation*, *replica deletion* and *objects migration*.

- *New Replica Creation*: insert some new replicas of database objects originally stored in a data partition into an another one containing no other replicas of the same objects.
- *Replica Deletion*: for database objects with multiple replicas, delete the specific replica within one partition.
- *Objects Migration*: relocate some database objects between two partitions; the procedure is realized by first inserting new replicas of them into the destination partition and then deleting the original ones from the source partition.

To execute the repartition operations, the scheduler packages them into repartition transactions using the information provided by the optimizer. The repartition transaction will be scheduled by the repartitioner and submitted to the system at a chosen time. It utilizes a cost-based approach to determine the repartition transactions' execution orders, and leverages a feedback model in control theory to determine at which frequency the repartition transactions should be scheduled for execution. As such, the processing of repartition transactions and normal transactions may be interleaved. In other words, during the database repartitioning, the processing of normal transactions will keep going on, and an online scheduling algorithm, which will be elaborated in Section 3, attempts schedule repartition transaction so that the time frame of executing the repartition operations is minimized while guaranteeing the correctness and performance of the concurrent processing of normal transactions.

The repartitioner accesses the system logs, manipulates the processing queue and updates the mapping information and routing rules in the query router during and after the database repartitioning. With the piggyback-based execution method, the repartitioner may need to modify the normal transactions by inserting additional repartition operations to some of them, and the transaction manager will coordinate the processing of the modified normal transactions.

3. ONLINE REPARTITION SCHEDULING

The scheduling of repartition operations has to be done in an online fashion. Besides the incoming workload is hard to predict, there are many system factors that will cause the system performance to fluctuate over time, such as variations of network speeds/bandwidth, transaction failures, and interferences from other programs running on the same server. Therefore, we study how to implement an online scheduler that can continuously adapt to the system's actual workload and capacity.

3.1 Generating and Ranking Repartition Transactions

In all the subsequent scheduling algorithms, we have to first decide the execution order of repartition transactions and schedule the more "beneficial" ones before those less "beneficial". To achieve this, we need to estimate the cost and benefit of executing such transactions. To estimate the cost of a transaction under different partitioning plans, we follow the approach in [4]. Suppose the cost of running transaction T_i with a repartition plan where all the tuples accessed by T_i are collocated in a single partition is C_i, then the cost of T_i with a plan where T_i has to access more than one partition is $2C_i$.

To estimate the benefit of a repartition transaction, we use the cost model of the data partitioning algorithms, such as [4, 13, 15]. Suppose the cost of an arbitrary normal transaction T_i with partition plan $\mathcal{P}$ is $C_i(\mathcal{P})$, then the benefit of a repartition transaction T_j, denoted as B_j can be defined as $\Sigma_{\forall T_i} f_i(C_i(\mathcal{O}) - C_i(\mathcal{P}))$, where f_i is the frequency of T_i. Finally, we can define the benefit density of T_j as B_j/C_j and then we can schedule the repartition transactions in descending order of their benefit densities.

To package the repartition operations into transactions, there are two simple options: (1) putting all operations into one transaction and (2) creating one transaction for each operation. The first option will create a very large transaction especially when there are a lot of data to be repartitioned. Such a large transaction will be run for a very long time and will significantly increase resource contention. For example, with a 2PL policy, the repartition transaction has to hold the locks of all the data objects involved in the repartition plan until it is committed. This will substantially increase the degree of lock contention with the normal transactions. On the other hand, the second option will not suffer from this problem but it will create a lot of transactions each incurring overhead to the transaction manager. It is desirable to find a good trade-off between this two extremes. In principle, we would like to create small transactions and their overhead can be paid off by the benefit that it will bring to the system. As accurately predicting and quantifying the overhead and the degree of lock contentions that will be introduced by a repartition transaction is difficult, we adopt a simple heuristic here. Roughly speaking, we put the repartition operations that repartition all the objects accessed by a normal transaction into a repartition transaction. In this way, we can ensure that there is at least one normal transaction that will benefit from executing the repartition transaction. Provided that the achieved benefit is greater than the overhead of introducing the repartition transaction, we can ensure that the overhead will be paid off when the repartition transaction is executed. Furthermore, even with this heuristic, there are still many possible ways to combine the repartition operations into transactions. We prefer generating transactions that will have higher benefit densities and, as mentioned earlier, schedule them in descending order of their benefit densities.

Algorithm 1 shows the whole process for the scheduler to generate a ranked list of repartition transactions. Given a new partition plan $\mathcal{P}$ generated by a cost-based repartition optimizer, the scheduler will obtain a list of repartition operations OP_{rep} together with the list of normal transactions that will access the data objects modified by each operation. In line 1-8, we construct a map T_{OP} that maps the ID of a normal transaction t_i with frequency f_i to a group of repartition operations that will modify objects accessed by t_i. In other words, the performance of t_i will be affected by this list of repartition operations.

We then calculate the benefit of executing each repartition operation in lines 9-14. After that, we can calculate the total benefits of each group of repartition operations in T_{op} and store them in another map $T_{benefit}$ with value descending order. Finally, we transform each group of repartition operations into a repartition transaction, and make sure each repartition operation only belongs to one repartition transaction. These repartition transactions will be returned by the algorithm as the output. Furthermore, we calculate the benefit density of each repartition transaction and sort them in descending order. Given a repartition transaction r_i and a normal transaction t_i whose performance will be affected by r_i, T_{Rep} maps the ID of r_i to the ID of t_i and the benefit density of r_i. Such auxiliary information will be used in our subsequent scheduling algorithms.

Algorithm 1: Generating and Ranking Repartition Transactions

```
Data: a list of repartition operations $OP_{rep}$ generated
      by optimizer, new partition plan $\mathcal{P}$
Result: a list of repartition transactions $L_{Rep}$,a map
        $T_{Rep}$ mapping repartition transaction id to a
        affected normal transaction id and the benefit
        density of the repartition transaction
1  Create HashMap $T_{op}$, a mapping from normal
   transaction to the repartition operations that edit the
   objects visited by it
2  for $op_k \in OP_{rep}$ do
3      for Normal transaction $t_i$ accessing the objects
       modified by $op_k$ do
4          if $C_i(\mathcal{O}) - C_i(\mathcal{P}) > 0$ then
5              Insert $op_k$ to $T_{op}$.get($t_i$)
6  for $t_i \in T_{op}.keylist$ do
7      benefit $\leftarrow f_i \frac{C_i(\mathcal{O})-C_i(\mathcal{P})}{T_{op}.get(t_i).size()}$
8      for $op_k \in T_{op}.get(t_i)$ do
9          $op_k.benefit$ += $benefit$
10 Create HashMap $T_{benefit}$, a mapping from repartition
   operation group ID to the total benefit for system if all
   the operations within this group are executed
11 for $(t_i, L_{op}) \in T_{op}.entrySet$ do
12     benefit $\leftarrow$ 0; for $op_i \in L_{op}$ do
13         benefit += $op_i.benefit$;
14     Insert ($t_i$,benefit) to $T_{benefit}$;
15 Sort $T_{benefit}$ with value descending order
16 for $(t_i, benefit) \in T_{benefit}$ do
17     ops $\leftarrow T_{op}.get(t_i)$;
18     for $op_i \in ops$ do
19         if $op_i \notin OP_{rep}$ then
20             Remove $op_i$ from ops; $T_{benefit}.get(t_i) \leftarrow$
               $T_{benefit}.get(t_i) - op_i.benefit$;
21     Remove ops from $OP_{rep}$;
22     Create $r_i$ with ops;
23     $c_i \leftarrow$ Cost($r_i$,$\mathcal{O}$);
24     $cpr_i \leftarrow \frac{T_{benefit}.get(t_i)}{c_i}$;
25     Insert $((r_i, t_i), cpr_i)$ to $T_{Rep}$;
26     Insert $r_i$ to $L_{Rep}$
27 Sort $T_{Rep}$ with value descending order;
```

3.2 Basic Solution

In general, there is a tension between the two objectives in our scheduling: (1) executing the repartitioning queries as soon as possible to improve the current partitioning plan, (2) avoiding interferences to the normal transactions and making the repartition process transparent to the end users. In this subsection, we propose two baseline solutions, each favoring one of the objectives.

Apply-All. This strategy is to maximize the speed of applying the repartitioning plan and submits all the repartition transactions to the waiting queue with a priority higher than the normal transactions. As mentioned earlier, the system will schedule the transactions in descending order of their priorities and hence this strategy is equivalent to pausing the processing of normal transactions and performing the repartitioning queries immediately. Depending on the number of repartition transactions, the normal transactions may need to wait for a rather long time, which is usually unacceptable.

After-All. To minimize the interferences to normal transactions, we can use a lazy strategy where repartition transactions will only be scheduled when the system is idle. We can achieve this by giving all the repartition transactions a priority lower than the normal ones. By doing so, the normal transactions will almost not be affected by repartition transactions and the repartitioning could be done transparently. Due to this advantage, a state-of-the-art approach for online repartitioning adopted this strategy [15]. However, there is a downside of this approach: the repartitioning may be per-

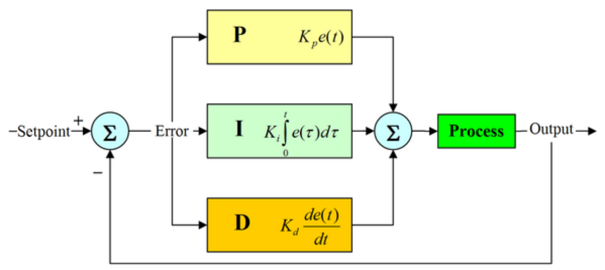


Figure 2: A sample PID controller block diagram

formed too slowly, especially when the system workload is high and there is very little idle time. Under this situation, the high workload could actually be alleviated by adopting the new and better partitioning plan and this strategy fails to take advantage of those.

3.3 Feedback-based Approach

As discussed earlier, the aforementioned basic solutions lack the flexibility to find a good trade-off between the two contradicting objectives. To achieve this, one can schedule some additional repartition transactions on top of those scheduled by the After-All strategy. These additional transactions will be assigned with the same priority as the normal transactions so that they have the chance to be executed faster. We call such transactions as high-priority repartition transactions to distinguish them with those low-priority ones scheduled by the After-All strategy. To limit the impact over the normal transactions, we can limit the number of high priority repartition transactions.

However, such a seemingly simple idea is rather challenging to realize in practice. Note that the number high-priority repartition transactions that we can execute without significant disturbance of the normal transactions heavily depends on the system's current workload and capacity. In reality, the system's workload may have temporal skewness and fluctuations even if it appears to be uniformly distributed for a long period [13]. Furthermore, the system's capacity is also subject to variations caused by external factors, such as external workload imposed on the same server or other virtual servers running on the same physical machine or cluster rack in a cloud computing environment. A desirable solution should be able to detect such short-term variations of system workload and capacity and promptly adapt the scheduling strategy accordingly. To achieve this goal, we model our system as an automatic control system and make use of the feedback control concept in control theory to design an adaptive scheduling strategy.

Control theory deals with the behaviors of complex dynamic systems with inputs and present output values. A controller is engineered to generate proper corrective actions so that system error, i.e. the differences between the desired output value, called setpoint (SP), and the actual measured output value, called process variable (PV), are minimized.

A commonly used controller is the Proportional-Integral-Derivative controller (PID controller). Figure 2 depicts a graphical representation of a PID controller. Let $u(t)$ be the output of the controller, then the PID controller can be defined as follows:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{d}{dt} e(t) \qquad (1)$$

where K_p, K_i and K_d are the proportional, integral and derivative gains respectively, and $e(t)$ is the system error at time t. The system error can be minimized by tuning the three gains so that the controller will generate proper outputs. Simply put, K_p, K_i and K_d determines how the present error ($e(t)$), the accumulation of past errors ($\int_0^t e(\tau)d\tau$) and the predicted future error ($\frac{d}{dt}e(t)$) would affect the controller's output.

The system for scheduling repartition transactions can be modeled as a PID controller as follows. We can use the ratio of the total cost of the high-priority repartition transactions to that of the normal transactions as the SP for the PID controller. By stabilizing this ratio, we can constrain the total workload imposed by the high-priority repartition transactions at a desirable level so that they would have limited impact over the latency of the normal transactions and in the mean time maximize the speed of applying the repartitioning plan.

To capture the fluctuations of the system's workload and capacity, we divide the time into small intervals and measure the aforementioned ratio for every interval. The actual ratio that is measured would be the PV of the PID controller and hence the error can be computed as $SP - PV$. The output of the controller is the ratio to be used to calculate the number of high-priority repartition transactions that we should schedule in the coming interval.

To tune the parameters of the PID controller, we take an online heuristic-based tuning method formally known as the Ziegler—Nichols method[19].

Finally, we enforce a limit on the maximum number of high-priority repartition transactions scheduled in each time interval to avoid significant impacts caused by sudden changes of system workload and capacity, which the PID controller will take some time to stabilize its outputs. Putting such a limit is essentially a conservative approach to avoid too much interferences during the period that the PID controller is stabilizing its behavior.

3.4 Piggyback-based Approach

In the feedback-based method the repartition transactions have the same priority as the normal transactions, hence they will contend with normal transactions for the locks of database objects and significantly increase the system's workload.

In this section, we propose a piggyback-based approach, which injects repartition operations into the normal transactions that access the same object. As these transactions would acquire the locks of these objects anyway, we can save the overhead of acquiring and releasing locks as well as performing the distributed commit protocols. Moreover, we can reduce the degree of lock contention by reducing the number of transactions.

The algorithm of this approach is illustrated in Algorithm 2. The algorithm make use of the auxiliary information produced in Algorithm 1. It examines the transaction ID t_i of the incoming normal transaction and check if there exist an repartition transaction r_j in T_{Rep} which t_i can benefit from but are not yet executed. If so, r_j will piggyback onto t_i, injecting the repartition operations in r_j to t_i. These

Algorithm	Workload	HighLoad			LowLoad		
		$\alpha = 100\%$	$\alpha = 60\%$	$\alpha = 20\%$	$\alpha = 100\%$	$\alpha = 60\%$	$\alpha = 20\%$
Feedback	Zipf	1.05	1.05	1.1	1.05	1.03	1.015
	Uniform	1.25	1.25	1.25	1.02	1.03	1.02
Hybrid	Zipf	1.05	1.05	1.05	1.05	1.03	1.05
	Uniform	1.05	1.05	1.05	1.03	1.05	1.05

Table 1: SP value for Experiments

Algorithm 2: Piggyback Algorithms for incoming normal transactions $T_i(k)$

```
Data: a list of repartition transactions $L_{Rep}$, a map
      $T_{Rep}$ from repartition transaction id to an
      affected normal transaction and the benefit
      density of the repartition transaction, incoming
      normal transactions $T_i(k)$ in interval $k$, List of
      all the repartition operations $OP_{Rep}$
Result: Piggybacked normal transaction executed in
        interval $k$
1  Create a map P(k) of all the normal transactions
   piggyback some repartition operations in interval $k$
2  for $t_i \in T_i(k)$ do
3      if $r_j, t_i \in T_{Rep}.keylist$ then
4          ops ← $L_{Rep}.get(r_j)$
5          Insert ops to $t_i$
6          Inert $(t_i, (r_j, t_i))$ to P(k)
7  Submit $T_i(k)$
8  Get Result(k) for any finished transaction
9  for $(t_i \in P(k))$ do
10     if $t_i \in Result(k)$ then
11         if $t_i.success$ then
12             Remove $P(k).get(t_i)$ from $T_{Rep}$
13         else
14             Remove $L_{Rep}.get(P(k).get(t_i).getKey())$
               from $t_i$
15             Resubmit $t_i$
```

repartition operations will share the locks of objects with the normal transactions. This essentially leads to a repartition-on-demand strategy where data will be repartitioned only when they are accessed. After the piggybacked transaction is successfully committed, we will remove the corresponding repartition transaction in T_{Rep}.

The piggyback method will increase the transaction failure rate as the execution times of normal transactions are increased. If too many repartition operations piggyback onto a normal transaction, then the system throughput will be decreased due to unnecessary aborts caused by the failure of the piggybacked repartition operations. Therefore, we need to limit the maximum number of repartition operations that can piggyback onto each normal transaction. This parameter should be tuned at runtime to adapt to the scenarios of different systems.

3.5 Hybrid Approach

While the piggyback-based approach consumes less resources, it fails to take use of the available system resources to speed up the repartitioning process. This may not work well when the system workload is low and there are few transactions to piggyback on. A more desirable approach is, when the system workload is low, we can take use of the available resources to repartition the data before the actual arrival of transactions that will access them, and when the system is a bit congested, we can take advantage of the opportunities to piggyback the repartition operations on the incoming transactions.

In this section, we present a hybrid approach that is composed by a piggyback module and the feedback module. The piggyback module will piggyback the repartition operations on the incoming normal transactions. Then for each interval, the feedback module will measure the actual PV value by counting in both the repartition transactions and those repartition operations piggybacked on the normal transactions. In this way, the feedback module will adapt the number of repartition transactions according to the actual workload of the system. In other words, when there are more incoming normal transactions that more repartition operations can piggyback on, we will reduce the number of repartition transactions and vice versa.

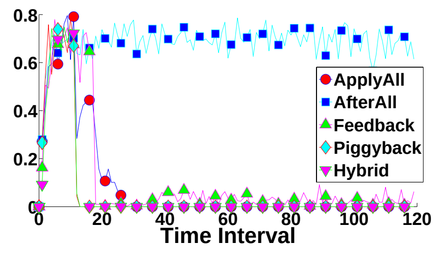
4. EVALUATION

In this section, we will first provide some details of our system implementation and experimental configuration in Section 4.1. The experimental results under different workload conditions are presented and discussed in Section 4.3 and Section 4.2.

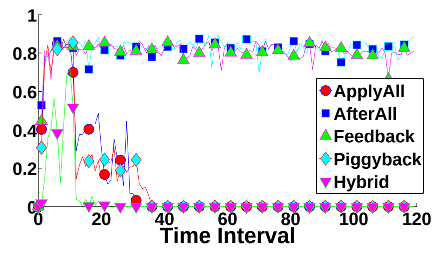
4.1 Experimental Configuration

System Implementation and Configuration. We have used PostgreSQL 9.2.4 as the local DBMS system at each data node and JavaSE-1.6 platform for developing and testing our algorithms. We have developed a query router using a lookup table to route each query to its target database objects. We have also implemented a query parser that reads a query and extracts the partition attributes of the target objects, which will be used for query routing and applying our online repartition strategies. For transaction management, we take use of Bitronix[17], an implementation of Java Transaction API 1.1 version adopting the XAResource interface to interact with the DBMS resource managers running each the individual data node and using 2-Phase Commit protocol for distributed transaction commits.

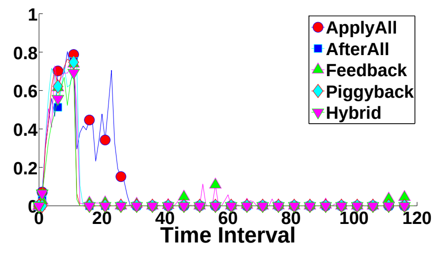
Our evaluation platform is deployed on a Amazon EC2 cluster consisting of 5 data nodes corresponding to 5 data partitions respectively. Each data node runs an instance of a PostgreSQL 9.2.4 server, which is configured to use the *read committed* isolation level and has a limitation of 100 simultaneous connections. Note that higher isolation level will decrease the system concurrency and hence lower the system's capacity. But it will not affect the performance of our algorithms. The node configuration consists of 1 vCPU using Intel Xeon E5-2670 processor and 3.75 GB memory with an on-demand SSD local storage running 64-bit Ubuntu



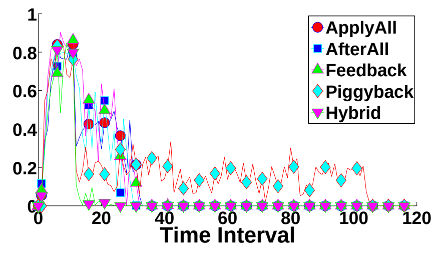
(a) Zipf/High Workload



(b) Uniform/High Workload



(c) Zipf/Low Workload



(d) Uniform/Low Workload

Figure 3: Experiment Results for Transaction Failure Rate

13.04. The query router is run on another EC2 instance with the same setting.

Workloads and Datasets. We create a table containing $500,000$ tuples, each tuple has a global unique key field and an integer content field. The size of each tuple is 8 bytes. We generate two types of workload distribution to simulate different sceneries of transaction popularity: a uniform distribution with $30,000$ distinct transactions and a Zipf distribution with $23,457$ distinct transactions. We generate the workload with a Zipf distribution using the parameter $s = 1.16$ so that the workload follows the 80-20 rule. Each normal transaction contains 5 queries. Each query access one unique tuple and is either a read-only or a write query with equal possibility.

We use a Poisson distribution to determine how many normal transactions are submitted to the system during each interval, which is set to be 20 seconds. Each run of the experiment will last for 45 minutes and the normal transactions are submitted to the system at the beginning of each time interval. We generate a high and a low workload as follows. *Lowload* has an average system utilization as 65% before the repartitioning, which is measured by the percentage of time that the system spend on processing the normal transactions. *Highload* simulates a system overloaded situation, where the incoming workload is 30% higher than the system capacity. Under this situation, it is more urgent to adopt the repartitioning plan to reduce the effective incoming load. Furthermore, for each situation, we vary the percentage of tuples α we need to repartition, which varies from 100% to 20%. After the repartitioning, α percent of the normal transactions would be tranformed from distributed transactions to non-distributed transactions.

Algorithm Settings. We compare all the algorithms we discussed in the previous sections. We use two performance metrics for comparison: the system throughput, which is counted as the maximum number of normal transactions that the system can process per unit of time, and the processing latency, which is the time between a transaction is submitted and the time its processing is finished. In order to examine the lock contentions incurred by the various scheduling algorithms, we also collect the failure rate of transactions, which is defined as the number of aborted transactions compared to the total number of transaction submitted to transaction manager.

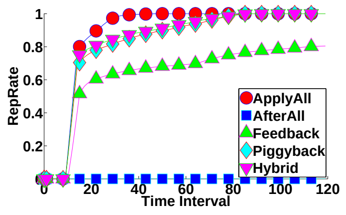
In line with the workload generation, we divide the time into 20 seconds of intervals and run the system for 10 intervals to warm it up before we start the repartitioning. Furthermore, the feedback-based approach uses 20 seconds as the monitoring interval. For the feedback model parameter used in each of the experiment, we have the different SP values which are listed in Table 1. All the experiments will have the same controller parameter $K_p = 1$, $K_i = 0$ and $K_d = 0$.

4.2 Performance Under High Load

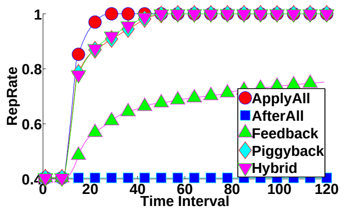
Recall that under the high workload setting, we have set the initial workload to be higher than the system's capacity but it should become lower than the system's capacity after applying the repartition plan as the normal transactions would consume less resources with the new partition plan. Therefore, it is necessary for the system to be able to process the repartition transactions soon.

Zipf workload. The experimental results are presented in Figure 4. As we discussed earlier, ApplyAll would stall all the normal transactions and execute all the repartition transactions before we resume the normal processing. This should result in the fastest deployment of the new partition plan. This is verified by Figures 4a, 4b and 4c. However as one can see from Figures 4d, 4e and 4f and Figures 4g, 4h and 4i, using this approach will experience a period that system has a very low throughput and very high processing latency caused by the stalling of the normal transactions.

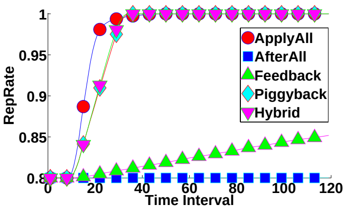
As shown in Figure 4i, the impact on processing latency can actually last much longer than the time needed to perform the repartitioning. This is because a long waiting queue



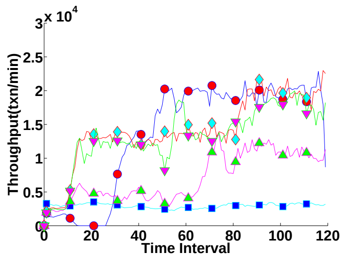
(a) $\alpha = 100\%$



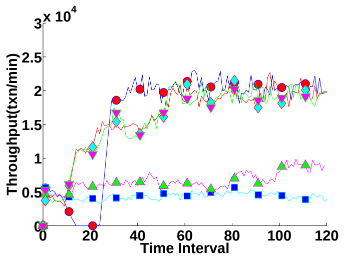
(b) $\alpha = 60\%$



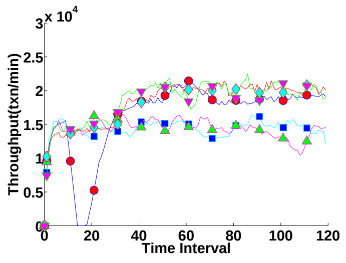
(c) $\alpha = 20\%$



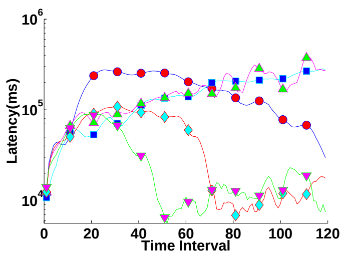
(d) $\alpha = 100\%$



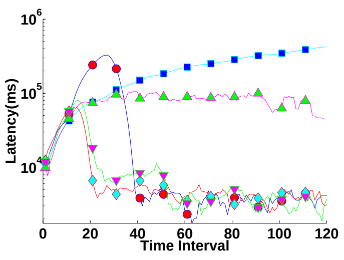
(e) $\alpha = 60\%$



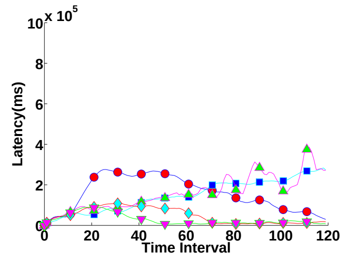
(f) $\alpha = 20\%$



(g) $\alpha = 100\%$



(h) $\alpha = 60\%$



(i) $\alpha = 20\%$

Figure 4: Experiment Results for Zipf High Workload

will be built up during the repartitioning process and hence it needs a longer time to clear the queue. (Note that we have set the initial ratio of workload to system capacity be roughly equal for all α values and we actually submit more normal transactions in the case with a lower α value. Hence the waiting queue will be longer in this case.)

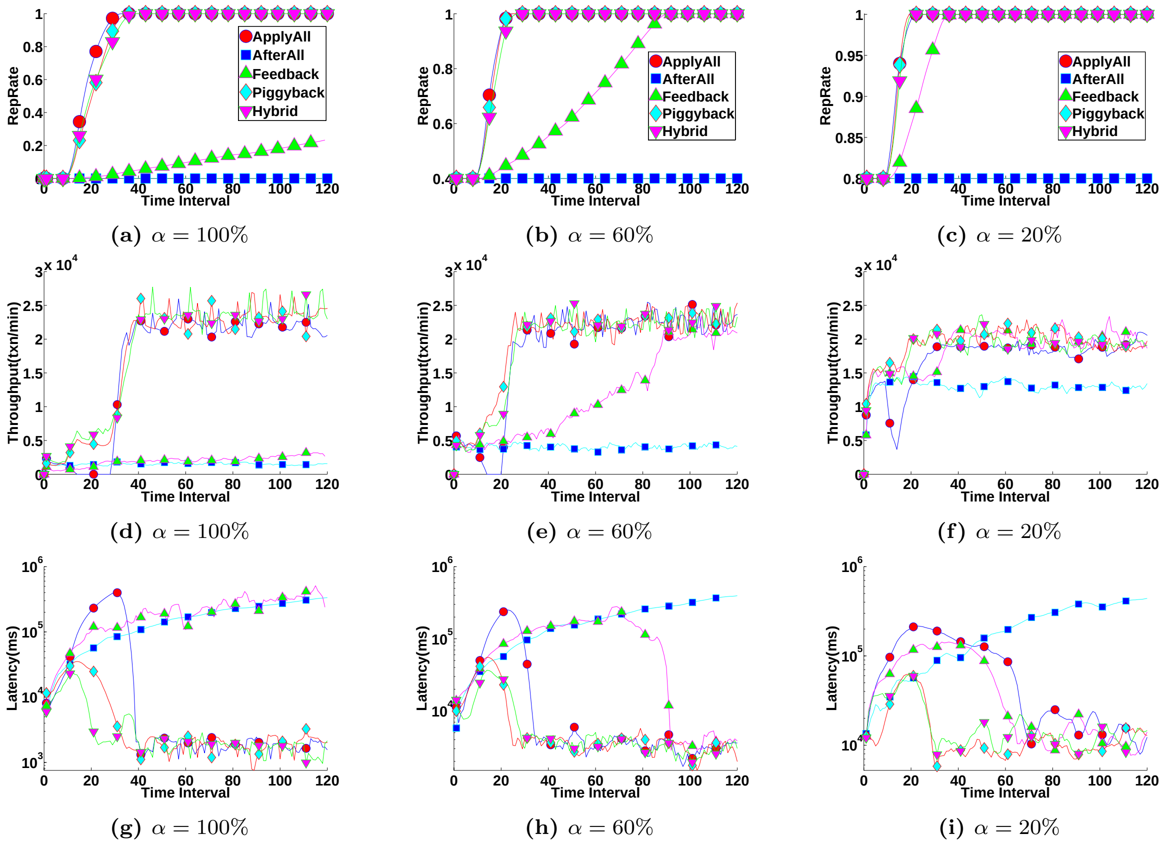
On the contrary, AfterAll basically cannot execute any repartition transactions due to the lack of system idle time, hence it cannot take advantage of the new data partition plan. The Feedback approach will enforce the scheduling of some repartition transactions, hence can make some progress in deploying the new partition plan (Figures 4a, 4b and 4c). Accordingly the system's throughput and processing latency will improve gradually. (Again, we submit more normal transactions in the case with a lower α value. So it takes a longer time to deploy the repartition plan.)

As we analyzed in the earlier sections, the Piggyback approach can effectively reduce the cost of executing repartition operations. This is especially important when the system is under high workload and has little extra resources for repartitioning the data. Furthermore, the high arrival rate of normal transactions provides abundant opportunities for the repartition operations to piggyback. The results in Figure 4 verify our analysis. In comparing to ApplyAll, both Piggyback and Hybrid do not incur any sudden dropping of system performance while is able to quickly execute the repartition plan. It even outperforms ApplyAll at almost all time intervals in terms of both throughput and latency with a lower α value, i.e. fewer tuples to be repartitioned.

Uniform workload. We also perform the experiments with a workload under a uniform distribution. The results are presented in Figure 5. The difference from the workload with a Zipf distribution is that we will not gain a lot of improvement by executing a small portion of repartition transactions.

Similar to the previous experiments, since the workload is more than the system could handle, AfterAll could barely execute any repartition transactions improve the system's performance, while ApplyAll finish the repartition process in 20,12 and 4 intervals, which is proportional to the number of repartition transactions that need to be executed.

For the Feedback method, we set a higher SP value under uniform workload to examine its performance when more repartition transactions are enforced to be submitted to the system. Under $\alpha = 100\%$, we cannot apply enough repartition transactions to stop the queue size from increasing. So even the repartition rate increases a bit in figure 5a, the throughput and latency does not get improved. But under the cases with $\alpha = 60\%$ and $\alpha = 20\%$, since the number of repartition transactions we need to execute is smaller, the system finally finish the repartition in time and make the system able to process all the incoming normal transactions without queuing. In comparing the results in the previous experiments, a higher SP here is actually beneficial when the number of repartitioning transactions is relatively small and Feedback has the chance to finish them in a good time.



(a) $\alpha = 100\%$ **(b)** $\alpha = 60\%$ **(c)** $\alpha = 20\%$

(d) $\alpha = 100\%$ **(e)** $\alpha = 60\%$ **(f)** $\alpha = 20\%$

(g) $\alpha = 100\%$ **(h)** $\alpha = 60\%$ **(i)** $\alpha = 20\%$

Figure 5: Experiment Results for Uniform High Workload

Similar to the Zipf workload, both Piggyback and Hybrid performed the best in all cases. They can achieve a speed that are almost identical to the *ApplyAll* approach.

Transaction failure rate. To further investigate the effect of the algorithms, we also collect the failure rates of transactions. Here we only report the results with $\alpha = 100\%$ since we could experience highest lock contention in these scenarios. The results are shown in Figure 3. We can see from Figure 3a, AfterAll has a very high failure rate during the whole period simply because the system's workload is very high and it fails to apply the repartition plan to quickly improve the system's performance. Furthermore, both the piggyback and hybrid method has a very low failure rate through the whole period, which clearly show the piggyback-based method's advantage of lowering lock contentions. On the other hand, the feedback-based method experiences some failure caused by the extra transactions scheduled by the feedback-based method.

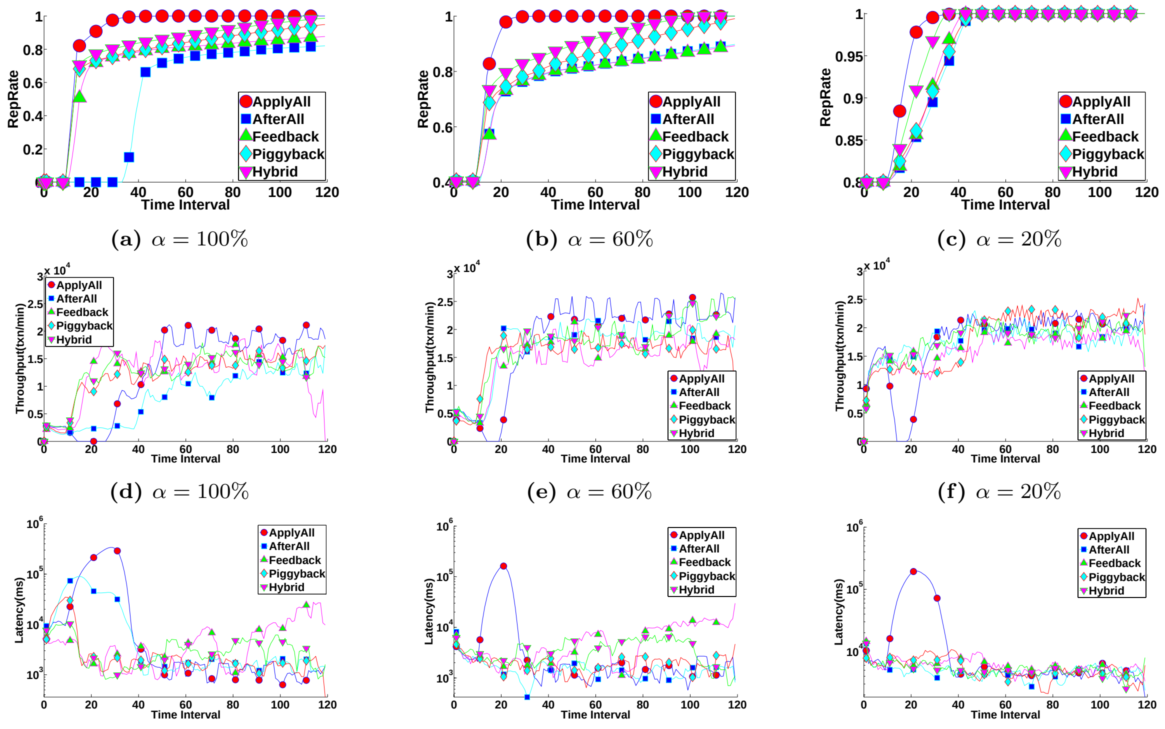
Figure 3b shows the results with the uniform workload. The general trend is similar. But it is interesting to see that the extra failure rate caused by the piggyback strategy lasts longer than the case with Zipf workload. This is because there is not any very hot transaction in this scenario, so we cannot deploy the repartition plan as quickly as in the case with Zipf workload. The mechanism of executing more high-priority repartition transactions with Hybrid causes a higher initial failure rate but it drops more quickly than the piggyback approach. This shows that Hybrid has an edge when there are less transactions to piggyback on.

4.3 Performance Under Low Load

In the low load experiments, we expect the system has more idle time and the repartition process could be done more aggressively to make use of those available resources. For the Zipf workload, since there exist some transactions that have very high frequencies, the resource contention under the same load level will be higher than the Uniform workload. The results are shown in Figure 6 and 7.

Zipf workload. ApplyAll performs similarly as under high load situation. But since there are fewer normal transactions, there are also fewer transactions that are queued up during the repartitioning period and we have a shorter time for the system to achieve its maximum performance after the repartitioning period.

As the system has enough idle time now, AfterAll could submit quite some repartition transactions. In Figure 6a, we can see that it takes some time for the system to have an idle period. It happens that there are three intervals that have a very high workload generated by our Poisson load generator. When the tuples accessed by the high frequency normal transactions are not repartitioned yet, their average execution time will be much higher than normal because of resources contention. We could see that AfterAll does not have this problem in Figures 4b and 4c. AfterAll has the minimum interference to the normal transactions when the system is able to handle the normal transaction load like



(a) $\alpha = 100\%$ (b) $\alpha = 60\%$ (c) $\alpha = 20\%$

(d) $\alpha = 100\%$ (e) $\alpha = 60\%$ (f) $\alpha = 20\%$

(g) $\alpha = 100\%$ (h) $\alpha = 60\%$ (i) $\alpha = 20\%$

Figure 6: Experiment Results for Zipf Low Workload

the situations in Figure 6h and 6i and hence it could be the algorithm that can achieve the lowest average latency.

The Feedback method will add more repartition transactions to the system besides filling the idle period. So we can see in Figure 6g that it partitions the tuples accessed by some high frequency transactions and render the system load decreasing more quickly than AfterAll. Adding the extra repartition transactions will increase processing latency of the normal transactions. This extra latency is a trade-off against the repartitioning speed. In Figure 5d, we can see that Feedback has a higher throughput than AfterAll but with a higher latency before it finishes all the repartition transactions.

The overhead of the piggyback method is proportional to the number of piggybacked transactions. When the workload is low, like the condition in Figure 6f, it may not be able to repartition the database as fast as the other methods. But the overhead on latency is also lower when the workload is low, which is similar to AfterAll.

Hybrid performs very well even under low workload. It always finishes the repartitioning work with a speed that is only slower than ApplyAll and in the meantime keeps the latency overhead less than the Feedback method. Since the repartitioning speed of Hybrid is faster than Feedback, the time period that normal transactions will experienced some extra latency are much shorter.

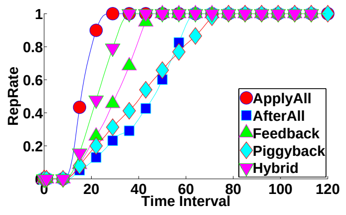
With regard to the failure rate, the trend shown in Figure 3c is very similar to the case with high workload, except that AfterAll has a much lower failure rate in this case. This is because the system's workload is not that high and AfterAll has the opportunity apply the repartition plan to further improve the system's performance.

Uniform workload. The results are reported in Figure 7. In this case, the degree of resource contention among normal transactions is lower than that with the Zipf workload. AfterAll could finish the repartitioning more quickly than with the Zipf load. Since the frequency of each normal transaction is low, the effect of repartition may take some time to get into effect. An interesting phenomenon is that Piggyback in this case works worse than the previous cases. With the uniform and low load situation, there are relatively few transactions to be piggybacked on and hence it take much longer for the piggyback approach to finish the repartitioning.

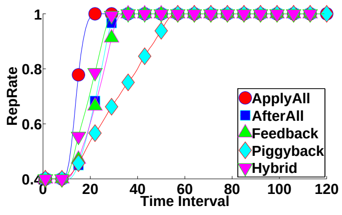
Furthermore, from Figure 3d, we can find that before the repartitioning is finished, the piggyback approach incurs a higher failure rate. This is because, even though Piggyback does not incur additional transactions, the piggybacked transactions will run longer than normal and this may still increase lock contentions to a certain degree. Given the longer repartitioning period in this case, its overhead on failure rate is more prominent here. On the contrary, Hybrid is able to make use of the available system resources to speed up the repartition and hence do not suffer this problem.

5. RELATED WORKS

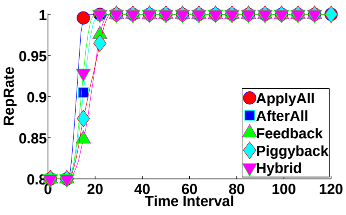
Data partitioning for distributed database system is about designing a data placement strategy minimizing the transaction overheads and balancing the workloads. Besides basic algorithms using some static functions, such as range-based or hash-based, to partition the data, researchers have recently been focusing on workload-aware partitioning al-



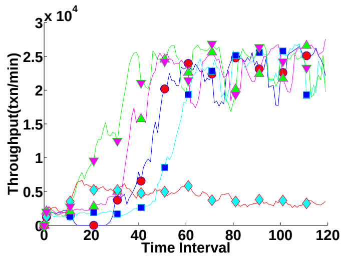
(a) $\alpha = 100\%$



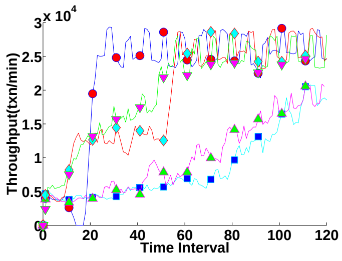
(b) $\alpha = 60\%$



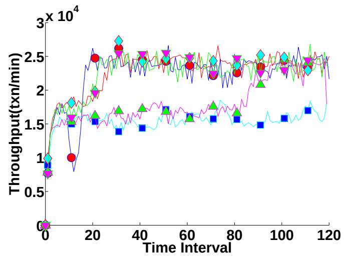
(c) $\alpha = 20\%$



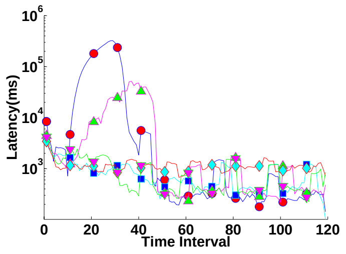
(d) $\alpha = 100\%$



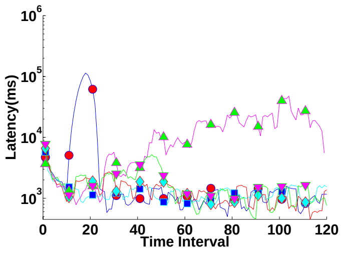
(e) $\alpha = 60\%$



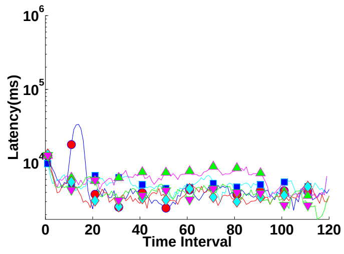
(f) $\alpha = 20\%$



(g) $\alpha = 100\%$



(h) $\alpha = 60\%$



(i) $\alpha = 20\%$

Figure 7: Experiment Results for Uniform Low Workload

gorithms that take the transaction statistics as input [4, 13]. These solutions utilize various techniques to model the historical workload information and search for an optimal partition scheme according to a specific objective function. Schism [4] is an automatic partitioning tool trying to minimize distributed transactions. Horticulture [13] further improve this approach by considering temporary skewness of workloads and using a local search algorithm to optimize partition schemes. This work provides a cost model for processing a transaction that considers both the number of partitions the transaction need to access and the overall skewness of data access.

Alekh etc.[8] present an online partitioning method that will partition the data in checkpoint time intervals. They generate partition schemes based on historical query execution logs and automatically update the partition plan when the workload changes. Besides the static partitioning methods, there are also some studies on incremental partitioning. For example, Sword [15] adopts a similar model as Schism [4] and introduces an incremental repartitioning algorithm that calculates the contribution of each repartition operation and the cost of executing it. They also propose a threshold on the number of repartition queries that will be generated for each repartition step and a constrained number of repartition queries will be executed when the system is at a lean period. This approach is similar to our baseline solution *AfterAll*. Some commercial database systems [12, 10] support online partitioning. But all of them require the partitioned data would not be untouched by normal transactions during the partitioning period. Hence this is similar to our *ApplyAll* solution. In our former short paper[1], we briefly presented the basic ideas about the feedback and piggyback algorithms. In this paper, we provide a more thorough analysis of the problem, consider the drawbacks of the piggyback solution and provide more experiment results to illustrate the trade-offs in the piggyback solution. We also propose the hybrid approach that combines the feedback and piggyback algorithm to benefit from the advantages of both algorithms while avoiding the problem of high failure rate in the piggyback solution and the problem of high interference with normal transactions in the feedback solution.

On the other hand, some researches on quality of service (QoS) of OLTP systems, such as [11], have considered the different resources' influence on transaction performance and have attempted to find the bottleneck resource for OLTP transactions and shows an arresting performance improvement. [6] makes use of machine learning methods to predict multiple performance metrics of analytical queries. The solution relies on SQL text extracted from the query execution plans. In [18], the authors addressed predictions of query execution time using the query optimizer's cost model and the generated query plan, which can be used to estimate the transaction execution time in OLTP systems. For distributed systems, the extra cost of network communication will be the new bottleneck, which limits the OLTP transactions performance [3]. It is an important part of execution cost if we want to optimize the transaction performance in a distributed environment.

Live migration of databases [5] focuses on migrating a whole database from one system to another while providing non-stop services and has not considered the scenario of migrating data from multiple nodes to multiple destinations, which however is the common case in our online data partitioning problem and may encounter distributed transactions that update the data at both the source and the destination nodes simultaneously. The authors of [5] provide a solution of combining on-demand pull and asynchronous push to migrate a tenant with minimal service interruption. Their solution is somehow similar to our piggyback approach, where data that needs to be moved will be migrated when the incoming transactions visit it.

Transaction scheduling is an important topic studied in various areas such as Web services and database systems, and there are several works, such as [7, 2]), that tried to find an optimal schedule by considering query execution time, transaction deadline and system workload. Given the transactions' execution time and hard execution deadlines, most of the scheduling problems are NP-Complete [16]. The most common solution is cost-based algorithms [9]. The quality of a schedule highly depends on the cost estimation and how the execution cost each transaction is modeled. [14] is a scheduling and admission control method using a priority token bank in computer networks. They classify jobs into N classes and jobs within each class are treated equally (by using FIFO). This approach is much simpler than cost-based scheduling (CBS).

6. CONCLUSION

In this paper, we studied the problem of online repartitioning of a distributed OLTP database. We identify that the two basic solutions are very rigid and miss the opportunities to find good trade-offs between the speed of repartitioning and the impact on the normal transactions. We then propose to use control theory to design an adaptive methods which can dynamically change the frequency that we submit repartition transactions to the system. As putting the repartition queries into extra transactions may further increase the system's resource contention especially when the system has a high workload, we also proposed a piggyback-based method to mitigate the repartitioning overhead, which however do not perform well when the system has a low workload and there is few transactions to piggyback on. Our hybrid approach intelligently integrates the two approaches and is able to combine their strengths while avoiding their problems. Based on the experiments of running our prototype on Amazon EC2, we can conclude that Hybrid is the overall best approach and achieves a great performance improvement in comparing to the two basic solutions used in most existing systems.

7. REFERENCES

[1] Kaiji Chen, Yongluan Zhou, and Yu Cao. Scheduling online repartitioning in oltp systems. In *Proceedings of the Middleware Industry Track*, Industry papers, pages 4:1–4:6, 2014.

[2] Yun Chi, Hakan Hacígümüş, Wang-Pin Hsiung, and Jeffrey F Naughton. Distribution-based query scheduling. *Proceedings of the VLDB Endowment*, 6(9):673–684, 2013.

[3] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I Jordan, and Ion Stoica. Managing data transfers in computer clusters with orchestra. 41:98–109, 2011.

[4] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment*, 3(1-2):48–57, 2010.

[5] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Zephyr: Live migration in shared nothing databases for elastic cloud platforms. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 301–312, 2011.

[6] Archana Ganapathi, Harumi Kuno, Umeshwar Dayal, Janet L Wiener, Armando Fox, Michael I Jordan, and David Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. pages 592–603, 2009.

[7] Shenoda Guirguis, Mohamed A Sharaf, Panos K Chrysanthis, Alexandros Labrinidis, and Kirk Pruhs. Adaptive scheduling of web transactions. pages 357–368, 2009.

[8] Alekh Jindal and Jens Dittrich. Relax and let the database do the partitioning online. In *BIRTE*, pages 65–80, 2011.

[9] Joseph YT Leung. *Handbook of scheduling: algorithms, models, and performance analysis*. CRC Press, 2004.

[10] MSDN Library. Partitioned tables and indexes.

[11] David T McWherter, Bianca Schroeder, Anastassia Ailamaki, and Mor Harchol-Balter. Priority mechanisms for oltp and transactional web applications. pages 535–546, 2004.

[12] Oracle. Using partitioning in an online transaction processing environment.

[13] Andrew Pavlo, Carlo Curino, and Stanley B. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman, editors, *SIGMOD Conference*, pages 61–72. ACM, 2012.

[14] Jon M Peha. Scheduling and admission control for integrated-services networks: the priority token bank. *Computer Networks*, 31(23):2559–2576, 1999.

[15] Abdul Quamar, K. Ashwin Kumar, and Amol Deshpande. Sword: Scalable workload-aware data placement for transactional workloads. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 430–441, 2013.

[16] Jeffrey D. Ullman. *NP*-complete scheduling problems. *Journal of Computer and System sciences*, 10(3):384–393, 1975.

[17] Brett Wooldridge. Bitronix jta transaction manager, 2013.

[18] Wentao Wu, Yun Chi, Shenghuo Zhu, Junichi Tatemura, H Hacigumus, and Jeffrey F Naughton. Predicting query execution time: Are optimizer cost models really unusable? pages 1081–1092, 2013.

[19] JG Ziegler and NB Nichols. Optimum settings for automatic controllers. *trans. ASME*, 64(11), 1942.

Chariots : A Scalable Shared Log for Data Management in Multi-Datacenter Cloud Environments

Faisal Nawab Vaibhav Arora Divyakant Agrawal Amr El Abbadi

Department of Computer Science, University of California, Santa Barbara, Santa Barbara, CA 93106
{nawab,vaibhavarora,agrawal,amr}@cs.ucsb.edu

ABSTRACT

Web-based applications face unprecedented workloads demanding the processing of a large number of events reaching to the millions per second. That is why developers are increasingly relying on *scalable cloud platforms* to implement cloud applications. Chariots exposes a shared log to be used by cloud applications. The log is essential for many tasks like bookkeeping, recovery, and debugging. Logs offer linearizability and simple append and read operations of immutable records to facilitate building complex systems like stream processors and transaction managers. As a cloud platform, Chariots offers fault-tolerance, persistence, and high-availability, transparently.

Current shared log infrastructures suffer from the bottleneck of serializing log records through a centralized server which limits the throughput to that of a single machine. We propose a novel distributed log store, called the *Fractal Log Store* (FLStore), that overcomes the bottleneck of a single-point of contention. FLStore maintains the log within the datacenter. We also propose Chariots, which provides multi-datacenter replication for shared logs. In it, FLStore is leveraged as the log store. Chariots maintains causal ordering of records in the log and has a scalable design that allows elastic expansion of resources.

1. INTRODUCTION

The explosive growth of web applications and the need to support millions of users make the process of developing web applications difficult. These applications need to support this increasing demand and in the same time they need to satisfy many requirements. Fault-tolerance, availability, and a low response time are some of these requirements. It is overwhelming for the developer to be responsible for ensuring all these requirements while scaling the application to millions of users. The process is error-prone and wastes a lot of efforts by reinventing the wheel for every application.

The cloud model of computing encourages the existence of unified platforms to provide an infrastructure that provides the guarantees needed by applications. Nowadays, such an infrastructure for compute and storage services is commonplace. For example, an application can request a key-value store service in the cloud. The store exposes an interface to the client and hides all the complexities required for its scalability and fault-tolerance. We envision that a variety of programming platforms will coexist in the cloud for the developer to utilize. A developer can use multiple platforms simultaneously according to the application's needs. A micro-blogging application for example might use a key-value store platform to persist the blogs and in the same time use a distributed processing platform to analyze the stream of blogs. The shared log, as we argue next, is an essential cloud platform in the developer's arsenal.

Manipulation of shared state by distributed applications is an error-prone process. It has been identified that using immutable state rather than directly modifying shared data can help alleviate some of the problems of distributed programming [1–3]. The *shared log* offers a way to share immutable state and accumulate changes to data, making it a suitable framework for cloud application development. Additionally, the shared log abstraction is familiar to developers. A simple interface of append and read operations can be utilized to build complex solutions. These characteristics allow the development of a wide-range of applications. Solutions that provide transactions, analytics, and stream processing can be easily built over a shared log. Implementing these tasks on a shared log makes reasoning about their correctness and behavior easier and rid the developer from thinking about scalability and fault-tolerance. Also, the log provides a trace of all application events providing a natural framework for tasks like debugging, auditing, checkpointing, and time travel. This inspired a lot of work in the literature to utilize shared logs for building systems such as transaction managers, geo-replicated key-value stores, and others [6, 11, 13, 27, 28, 30, 33].

Although appealing as a platform for diverse programming applications, shared log systems suffer from a single-point of contention problem. Assigning a log position to a record in the shared log must satisfy the uniqueness and order of each log position and consequent records should have no gaps in between. Many shared log solutions tackle this problem and try to increase the append throughput of the log by minimizing the amount of work done to append a record. The most notable work in this area is the CORFU protocol [7] built on flash chips that is used by Tango [8]. The CORFU protocol is driven by the clients and uses a *centralized sequencer* that assigns offsets to clients to be filled later. This takes the sequencer out of the data path and allows the append throughput to be more than a single machine's I/O bandwidth. However, it is still limited by the bandwidth of the sequencer. This bandwidth is suitable for small clusters but cannot be used to handle larger demands encountered by large-scale web applications.

We propose **FLStore**, a distributed deterministic shared log system that scales beyond the limitations of a single machine. FL-

Store consists of a group of *log maintainers* that mutually handle exclusive ranges of the log. Disjoint ranges of the log are handled independently by different log maintainers. FLStore ensures that all these tasks are independent by using a deterministic approach that assigns log positions to records as they are received by log maintainers. It removes the need for a centralized sequencer by avoiding log position pre-assignment. Rather, FLStore adopts a *post-assignment* approach where records are assigned log positions *after* they are received by the Log maintainers. FLStore handles the challenges that arise from this scheme. The first challenge is the existence of gaps in the log that occur when a log maintainer has advanced farther compared to other log maintainers. Another challenge is maintaining explicit order dependencies requested by the application developer.

Cloud applications are increasingly employing geo-replication to achieve higher availability and fault-tolerance. Records are generated at multiple datacenters and are potentially processed at multiple locations. This is true for applications that operate on shared data and need communication to other datacenters to make decisions. In addition, some applications process streams coming from different locations. An example is Google's Photon [4] which joins streams of clicks from different datacenters. Performing analytics also requires access to the data generated at multiple datacenters. Geo-replication poses additional challenges such as maintaining *exactly-once* semantics (ensure that an event is not processed more than once), automatic datacenter-level fault-tolerance, and handling un-ordered streams.

Chariots supports multiple datacenters by providing a global replicated shared log that contains all the records generated by all datacenters. The order of records in the log must be consistent. The ideal case is to have an identical order at all datacenters. However, it is shown by the CAP theorem [12, 16] that such a **consistency** guarantee *cannot* be achieved if we are to preserve **availability** and **partition-tolerance**. In this work we favor availability and partition-tolerance, as did many other works in different contexts [14, 15, 20, 23]. Here, we relax the guarantees on the order of records in the log. In relaxing the consistency guarantee, we seek the strongest guarantee that will allow us to preserve availability and partition-tolerance. We find, as other systems have [5, 10, 19, 23, 31], that causality [21] is a sufficiently strong guarantee fitting our criterion [24].

In this paper we propose a cloud platform that exposes a shared log to applications. This shared log is replicated to multiple datacenters for availability and fault tolerance. The objective of the platform's design is to achieve high performance and scalability by allowing seamless elasticity. Challenges in building such a platform are tackled, including handling component and whole datacenter failures, garbage collection, and gaps in the logs. We motivate the log as a framework for building cloud applications by designing three applications on top of the shared log platform. These applications are: (1) a key-value store that preserves causality across datacenters, (2) a stream processing applications that handles streams coming from multiple datacenters, and (3) a replicated data store that provides strongly consistent transactions [27].

The contributions of the paper are the following:

- A design of a scalable distributed *log storage*, FLStore, that overcomes the bottleneck of a single machine. This is done by adopting a post-assignment approach to assigning log positions.

- Chariots tackles the problem of scaling causally-ordered geo-replicated shared logs by incorporating a distributed log storage solution for each replica. An elastic design is built to allow scaling to datacenter-scale computation. This is the first work we are aware of that tackles the problem of *scaling geo-replicated shared logs through partitioning*.

Consistency	Partitioned	Replicated	systems
Strong	✓	✗	CORFU/Tango [7, 8] LogBase [33] RAMCloud [29] Blizzard [25] Ivy [26] Zebra [18] Hyder [11]
Strong	✗	✓	Megastore [6] Paxos-CP [30]
Causal	✗	✓	Message Futures [27] PRACTI [10] Bayou [32] Lazy Replication [19] Replicated Dictionary [36]
Causal	✓	✓	Chariots

Table 1: Comparison of different shared log services based on consistency guarantees, support of per-replica partitioning, and replication.

The paper proceeds as the following. We first present related work in Section 2. The system model and log interface follows in Section 3. We then present a set of use cases of Chariots in Section 4. These are data management and analytics applications. The detailed design of the log is then proposed in Sections 5 and 6. Results of the evaluations are provided in Section 7. We conclude in Section 8.

2. RELATED WORK

In this paper we propose a geo-replicated shared log service for data management called Chariots. Here we briefly survey related work. We focus on systems that manage shared logs. There exist an enormous amount of work on general distributed (partitioned) storage and geo-replication. Our focus on work tackling shared logs stems from the unique challenges that shared logs pose compared to general distributed storage and geo-replication. We provide a summary of shared log services for data management application in Table 1. In the remainder of this section we provide more details about these systems in addition to other related work that do not necessarily provide log interfaces. We conclude with a discussion of the comparison provided in Table 1.

2.1 Partitioned shared logs

Several systems explored extending shared logs as a distributed storage spanning multiple machines. Hyder [11] builds a multi-version log-structured database on a distributed shared log storage. A transaction executes optimistically on a snapshot of the database and broadcasts the record of changes to *all* servers and appends a record of changes to the distributed shared log. The servers then commit the transaction by looking for conflicts in the shared log in an intelligible manner. LogBase [33], which is similar to network filesystems like BlueSky [34], and RAMCloud [29] are also multi-version log-structured databases.

Corfu [7], used by Tango [8], attempts to increase the throughput of shared log storage by employing a sequencer. The sequencer's main function is to pre-assign log position ids for clients wishing to append to the log. This increases throughput by allowing more concurrency. However, the sequencer is still a bottleneck limiting the scalability of the system.

Distributed and networked filesystems also employ logs to share their state. Blizzard [25] proposes a shared log to expose a cloud block storage. Blizzard decouples ordering and durability requirements, which improves its performance. Ivy [26] is a distributed file system. A log is dedicated to each participant and is placed in a distributed hash table. Finding data requires consulting all logs but appending is done to the participant's log only. The Zebra file system [18] employs log striping across machines to increase throughput.

2.2 Replicated shared logs

Causal replication. Causal consistency for availability is utilized by various systems [10, 19, 19, 23, 31, 32]. Recently, COPS [23] proposes causal+ consistency that adds convergence as a requirement in addition to causal consistency. COPS design aims to increase the throughput of the system for geo-replicated environments. At each datacenter, data is partitioned among many machines to increase throughput. Chariots targets achieving high throughput similarly by scaling out. Chariots differs in that it exposes a log rather than a key-value store, which brings new design challenges. Logs have been utilized by various replication systems for data storage and communication. PRACTI [10] is a replication manager that provides partial replication, arbitrary consistency, and topology independence. Logs are used to exchange updates and invalidation information to ensure the storage maintains a causally consistent snapshot. Bayou [32] is similar to PRACTI. In Bayou, batches of updates are propagated between replicas. These batches have a start and end times. When a batch is received, the state rolls back to the start time, incorporate the batch, and then roll forward the existing batches that follows. Replicated Dictionary [36] replicates a log and maintains causal relations. It allows transitive log shipping and maintains information about the knowledge of other replicas. Lazy Replication [19] also maintains a log of updates ordered by their causal relations. The extent of knowledge of other replicas is also maintained.

Geo-replicated logs. Geo-replication of a shared log has been explored by few data management solutions. Google megastore [6] is a multi-datacenter transaction manager. Megastore commit transactions by contending for log positions using Paxos [22]. Paxos-CP [30] use the log in a similar way to megastore with some refinements to allow better performance. These two systems however, operate on a serial log. All clients contend to write to the head of the log, making it a single point of contention, which limits throughput. Message Futures [27] and Helios [28] are commit protocols for strongly consistent transactions on geo-replicated data. They build their transaction managers on top of a causally-ordered replicated log that is inspired from Replicated Dictionary [36].

2.3 Summary and comparison

The related works above that build shared logs for data management applications are summarized in Table 1. We display whether the system support partitioning and replication in addition to the guaranteed ordering consistency. Consistency is either strong, meaning that the order is either identical or serializable for replicated logs or totally ordered for non-replicated logs. A system is *partitioned* if the shared log spans more than one machine for each replica. Thus, if a system of five replicas consists of five machines, they are *not* partitioned. A system is *replicated* if the shared log has more than one *independent* copy.

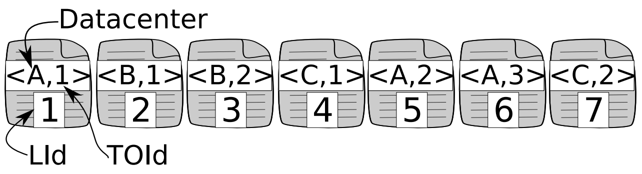


Figure 1: Records in a shared log showing their TOId inside the records alongside the datacenters that created them and the records LIds under the log

Other than Chariots, the table lists four systems that support partitioning. It is possible for these systems to employ replication in the storage level. However, a blind augmentation of a replication solution will be inefficient. This is because a general-purpose replication method will have guarantees stronger than what is needed to replicate a log. The other solutions, that support replication, do not support partitioning. Handling a replica with a single node limits the achievable throughput. The processing power, I/O, and communication of a single machine can not handle the requirements of todays web applications. This is specially true for geo-replication that handles datacenter-scale demand.

Chariots attempts to fill this void of shared logs that have both a native support of replication and per-replica partitioning. This need for both replication and partitioning has been explored for different applications and guarantees, including causal consistency, *i.e.*, COPS [23]. However, geo-replication of a distributed shared log and immutable updating pose unique challenges that are not faced by geo-replication of key-value stores and block-based storage. The paper studies these challenges and design Chariots as a causally-ordered shared log that supports per-replica partitioned log storage and geo-replication.

3. SYSTEM AND PROGRAMMING MODEL

Chariots is a shared log system for cloud applications. The inner workings of Chariots are not exposed to the application developer. Rather, the developer interacts with Chariots via a set of APIs. In this section, we will show the interface used by developers to write applications using Chariots.

System model. Chariots exposes a log of records to applications. The log is maintained by a group of machines called the *log maintainers*. Collectively, these log maintainers persist a single shared log. Each log maintainer is responsible for a disjoint range of the shared log. The shared log is accessed by cloud applications, called *application clients*, through a linked library that manages the exchange of information between the application and the log maintainers. Application clients are distributed and independent from one another. And they share a single view of the shared log. The shared log is fully replicated to a number of datacenters. In our model, we adopt a view of the *datacenter as a computer* [9], an increasingly popular view of computing that reflects the demand of datacenter-scale applications.

Meta information about log maintainers, other datacenters, and the shared log are managed by *meta servers*. Meta servers are a highly-available collection of stateless servers acting as an oracle for application clients to report about the state and locations of the Log maintainers and other datacenters. This model of scale-out distributed computing and centralized stateless highly-available control servers has been shown to perform the best for large-scale systems [17].

Data model. The state of the shared log consists of the records it contains. These records are either *local copies*, meaning that they were generated by application clients residing in the same datacenter, or *external copies*, meaning that they were generated at other datacenters. Each record has an identical copy at each datacenter, one of which is considered a local copy and the other copies are considered external copies. The record consists of the contents appended by the Application client, called the record's *body*, and other *meta-information* that are used by Application clients to facilitate future access to it. The following are the meta-information maintained for each record:

- *Log Id (LId):* This id reflects the position of the record in the datacenter where the copy resides. A record has multiple copies, one at each datacenter. Each copy has a different LId that reflects its position in the datacenter's shared log.
- *Total order Id (TOId):* This id reflects the total order of the record with respect to its host datacenter, where the Application client that created it resides. Thus, copies of the same record have an identical TOId.
- *Tags:* The Application client might choose to attach tags to the record. A tag consists of a key and a value. These tags are accessible by Chariots, unlike the record's body which is opaque to the system. Records will be indexed using these tags.

To highlight the difference between LId and TOId, observe the sample shared log in Figure 1. It displays seven records with their LIds in the bottom of each record at datacenter A. The TOId is shown inside the record via the representation $< X, i >$, where X is the host datacenter of the Application client that appended the record and i is the TOId. Each record has a LId that reflects its position in the shared log of datacenter A. Additionally, each record has a TOId that reflects its order compared to records coming from the same datacenter only.

Programming interface and model. Application clients can observe and change the state of the shared log through a simple interface of two basic operations: reading and appending records. These operations are performed via an API provided by a linked software library at the Application client. The library needs the information of the meta servers only to initiate the session. Once the session is ready, the application client may use the following library calls:

1. Append(in: record, tags): Insert record to the shared log with the desired tags. The assigned TOId and LId will be sent back to the Application client. Appended records are automatically replicated to other replicas.
2. Read(in: rules, out: records): Return the records that matches the input rules. A rule might involve TOIds, LIds, and tags information.

Log records are immutable, meaning that once a record is added, it cannot be modified. If an application client desire to alter the effect of a record it can do so by appending another record that exemplifies the desired change. This principle of accumulation of changes, represented by immutable data, is identified to reduce the problems arising from distributed programming [1–3]. Taking this principled approach and combining it with the simple interface of appends and reads allows the construction of complex software while reducing the risks of distributed programming. We showcase the potential of this simple programming model by constructing data management systems in the next section.

Causality and log order. The shared log at each datacenter consists of a collection of records added by application clients at different datacenters. Ordering the records by causality relations allows sufficient consistency while preserving availability and fault-tolerance [12, 16]. Causality enforces two types of order relations [21] between read and append operations, where $o_i \rightarrow o_j$ denotes that o_i has a causal relation to o_j. A causal relation, $o_i \rightarrow o_j$, exists in the following cases:

- **Total order** for records generated from the same datacenter. If two appended records, o_i and o_j, were generated by application clients residing in the same datacenter A, then if o_i is ordered before o_j in A, then this order must be honored at all other datacenters.
- **Happened-before relations** between read and append operations. A happened-before relation exists between an append operation, o_i, and a read operation, o_j, if o_j reads the record appended by o_i.
- **Transitivity:** causal relations are transitive. If a record o_k exists such that $o_i \rightarrow o_k$ and $o_k \rightarrow o_j$ then $o_i \rightarrow o_j$.

4. CASE STUDIES

The simple log interface was shown to enable building complex data management systems [6, 11, 13, 27, 30, 33]. These systems, however, operate on a serial log with pre-assigned log positions. These two characteristics, as we argued earlier, limits the log's availability and scalability. In this section, we demonstrate data management systems that are built on top of Chariots, a causally ordered log with post-assigned log positions. The first system, *Hyksos*, is a replicated key-value store that provides causal consistency with a facility to perform *get transactions*. The second system is a stream processor that operates on streams originating from multiple datacenters. We also refer to our earlier work, Message Futures [27] and Helios [28], which provide strongly consistent transactions on top of a causally ordered replicated log. Although they were introduced with a single machine per replica implementation, their design can be extended to be deployed on the scalable Chariots.

4.1 Hyksos: causally consistent key-value store

Hyksos is a key-value store built using Chariots to provide causal consistency [21]. Put and Get operations[1] are provided by Hyksos in addition to a facility to perform get transactions (GET_TXN) of multiple keys. Get transactions return a consistent state snapshot of the read keys.

4.1.1 Design and algorithms

Chariots manages the shared log and exposes a read and append interface to application clients, which are the drivers of the key-value store operations. Each datacenter runs an instance of Chariots. An instance of Chariots is comprised of a number of machines. Some of the machines are dedicated to store the shared log and others are used to deploy Chariots.

The value of keys reside in the shared log. A record holds one, or more put operation information. The order in the log reflects the causal order of put operations. Thus, the current value of a key, k, is in the record with the highest log position containing a put operation. The get and put operations are performed as follows:

[1]The terms "read" and "append" are used for operations on the log and "put" and "get" are used for operations on the key-value store.

Algorithm 1 Performing Get_transactions in Hyksos

```
1: // Request the head of the log position id
2: i = get_head_of_log()
3: // Read each item in the read set
4: for each k in read-set
5:     t = Read ({tag: k, LId<i}, most-recent)
6:     Output.add (t)
```

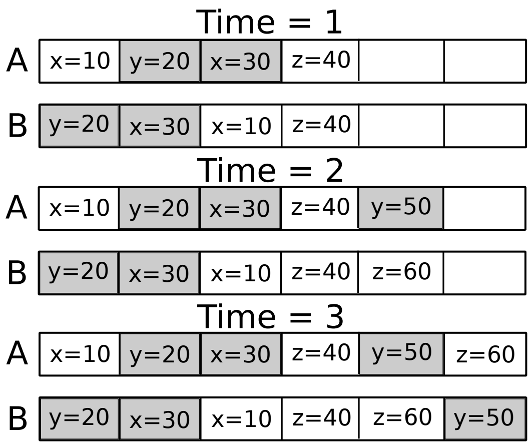


Figure 2: An example of Hyksos, the key-value store built using Chariots.

- Get(x): Perform a Read operation on the log. The read returns a recent record containing a put operation to *x*.
- Put(x, value): Putting a value is done by performing an Append operation with the new value of *x*. The record must be tagged with the key and value information to enable an efficient get operation.

Get transactions. Hyksos provides a facility to perform get transactions. The get_transaction operation returns a consistent view of the key-value store. The application client performs the Get operations as shown in Algorithm 1. First, Chariots is polled to get the head of the log's position id, *i*, to act as the log position when the consistent view will be taken (Line 2). There must be no gaps at any records prior to the log id. Afterwards, the application client begins reading each key *k* (Lines 4-6). A request to read the version of *k* at a log position *j* that satisfies the following: Record *j* contains the most recent write to *k* that is at a position less than *i*.

4.1.2 Example scenario

To demonstrate how Hyksos works, consider the scenario shown in Figure 2. It displays the shared logs of two datacenters, *A* and *B*. The shared log contains records of put operations. The put operation is in the form "$x = v$", where *x* is the key and *v* is the value. Records that are created by Application clients at *A* are shaded. Other records are created by application clients at *B*.

The scenario starts with four records, where each record has two copies, one at each datacenter. Two of these records are put operations to key *x*. The other two operations are a put to *y* and a put to *z*. The two puts to *x* were created at different datacenters. Note that the order of writes to *x* is different at *A* and *B*. This is permissible if no causal dependencies exist between them. At time 1, a Get of *x* at *A* will return 30, while 10 will be returned if the Get is performed at *B*.

At time 2, two Application clients, one at each datacenter, perform put operations. At *A*, Put(y,50) is appended to the log. At *B*, Put(z,60) is appended to the log. Now consider a get transaction that requests to get the value of *x*, *y* and *z*. First, a non-empty log position is chosen. Assume that the log position 4 is chosen. If the get transaction ran at *A*, it will return a snapshot of the view of the log up to log position 4. This will yield $x = 30$, $y = 20$, and $z = 40$. Note that although a more recent *y* value is available, it was not returned by the get transactions because it is not part of the view of records up to position 4. If the get transaction ran at *B*, it will return $x = 10$, $y = 20$, and $z = 40$.

Time 3 in the figure shows the result of the propagation of records between *A* and *B*. $Put(y,50)$ has a copy now at *B*, and $Put(z,60)$ has a copy at *A*.

4.2 Event processing

Another application targeted by Chariots is multi-datacenter event processing. Many applications generate a large footprint that they would like to process. The users' interactions and actions in a web application can be analyzed to generate business knowledge. These events range from click events to the duration spent in each page. Additionally, social networks exhibit more complex analytics of events related to user-generated contents (*e.g.*, micro-blogs) and user-user relationships to these events. Frequently, such analytics are carried in multiple datacenters for fault-tolerance and locality [4, 17].

Chariots enables a simple interface for these applications to manage the replication and persistence of these analytics while preserving the required exactly-once semantics. Event processing applications consist of publishers and readers. Publishing an events is as easy as performing an append to the log. Readers then read the events from the log maintainers. An important feature of Chariots is that readers can read from different log maintainers. This will allow distributing the analysis work without the need of a centralized dispatcher that can be a single-point of contention.

4.3 Message Futures and Helios

Message Futures [27] and Helios [28] are commit protocols that provide strongly consistent transactions on geo-replicated data stores. They leverage a replicated log that guarantees causal order [36]. A single node at each datacenter, call it replica, is responsible for committing transactions and replication. Transactions consist of read and write operations and are committed optimistically. Application clients read from the data store and buffer writes. After all operations are ready, a *commit request* is sent to the closest replica. A record is appended to the log to declare the transaction *t* as ready to begin the commit protocol. Message Futures and Helios implement different conflict detection protocols to commit transactions. Message Futures [27] waits for other datacenters to send their histories up to the point of *t*'s position in the log. Conflicts are detected between *t* and received transactions, and *t* commits if no conflicts are detected. Helios [28] builds on a lower-bound proof that determines the lowest possible commit latency that a strongly consistent transaction can achieve. Helios commits a transaction *t* by detecting conflicts with transactions in a *conflict zone* in the shared log. The conflict zone is calculated by Helios using the lower-bound numbers. If no conflicts were detected, *t* commits. A full description of Message Futures and Helios are available in previous publications [27, 28].

Message Futures and Helios demonstrate how a causally ordered log can be utilized to provide strongly consistent transactions on replicated data. However, the used replicated log solution [36] is rudimentary and is not suitable for today's applications. It only

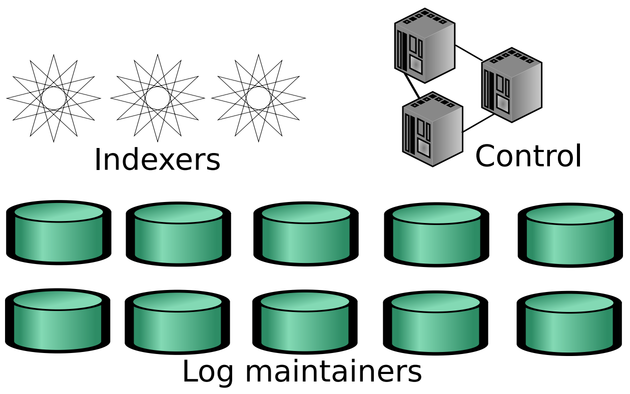


Figure 3: The architecture of FLStore

utilizes a single node per datacenter. This limits the throughput that can be achieved to that of a single node. Chariots can be leveraged to scale Message Futures and Helios to larger throughputs. Rather than a replica with a single node at each datacenter, Chariots would be used to distribute storage and computation.

4.4 Conclusion

The simple interface of Chariots enabled the design of web and analytics applications. The developer can focus on the logic of the data manager or stream processor without having to worry about the details of replication, fault-tolerance, and availability. In addition, the design of Chariots allows scalable designs of these solutions by having multiple sinks for reading and appending.

5. FLSTORE: DISTRIBUTED SHARED LOG

In this section we describe the distributed implementation of the shared log, called the Fractal Log Store (FLStore). FLStore is responsible of maintaining the log *within* the datacenter. We begin by describing the design of the distributed log storage. Then, we introduce the scalable indexing component used for accessing the shared log.

5.1 Architecture

In designing FLStore, we follow the principle of distributing computation and highly-available stateless control. This approach has been identified as the most suitable to scale out in cloud environments [17]. The architecture of FLStore consists of three types of machines, shown in Figure 3. *Log maintainers* are responsible for persisting the log's records and serving read requests. *Indexers* are responsible of access to log maintainers. Finally, control and meta-data management is the responsibility of a highly-available cluster called the *Controller*.

Application clients start their sessions by polling the Controller for information about the indexers and log maintainers. This information includes the addresses of the machines and the log ranges falling under their responsibility in addition to approximate information about the number of records in the shared log. Appends and reads are served by Log maintainers. The Application client communicates with the Controller only at the beginning of the session or if communication problems occur. And Application clients will communicate with Indexers only if read operation did not specify LIds in the rules.

5.2 Log maintainers

Scalability by post-assignment. The Log maintainers are accessed via a simple interface for adding to and reading from the shared log. They are designed to be *fully distributed* to overcome the I/O bandwidth constraints that are exhibited by current shared log protocols. A recent protocol is CORFU [7] that is limited by the I/O bandwidth of a sequencer. The sequencer is needed for CORFU to pre-assign log positions to application clients wishing to append records to the log. In FLStore, we abandon this requirement of pre-assigning log positions and settle for a *post-assignment* approach. The thesis of a post-assignment approach is to let the application client construct the record and send it to a randomly (or intelligibly) selected Log maintainer. The Log maintainer will assign the record the next available log position from log positions under its control.

Design. The shared log is distributed among the participating Log maintainers. This means that each machine holds a partial log and is responsible for its persistence and for answering requests to read its records. This distribution poses two challenges. The first is the way to append to the log while guaranteeing uniqueness and the non-existence of gaps in the log. This includes the access to these records and the way to index the records. The other challenge is maintaining explicit order guarantees requested by the application client. We employ a *deterministic* approach to make each machine responsible for specific ranges of the log. These ranges round-robin across machines where each round consists of a number of records. we will call this number the batch size. Figure 4 depicts an example of three log maintainers, *A*, *B*, and *C*. The figure shows the partial logs of the first three rounds if the batch size was set to a 1000 records.

If an application wants to read a record it directs the request to the Log maintainer responsible for it. The Log maintainer can only answer requests of records if their LIds are provided. Otherwise, the reader must collect the LIds first from the Indexers as we show in the next section. Appending to the log is done by simply sending a record or group of records to one of the Log maintainers. The Log maintainer appends the record to the next available log position. It is possible that a log maintainer will receive more record appends than others. This creates a load-balancing problem that can be solved by giving the application feedback about the rate of incoming requests at the maintainers. This feedback can be collected by the Controller and be delivered to the application clients as a part of the session initiation process. Nonetheless, this is an orthogonal problem that can be solved by existing solutions in the literature of load balancing.

5.3 Distributed indexing

Records in Log maintainers are arranged according to their LIds. However, Application clients often desire to access records accord-

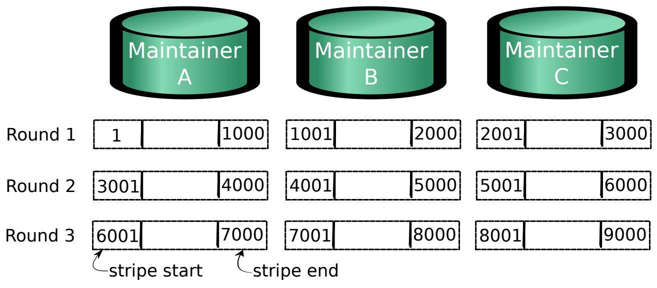


Figure 4: An example of three deterministic log maintainers with a batch size of 1000 record. Three rounds of records are shown.

ing to other information. When an Application client appends a record it also *tags* it with access information. These tags depend on the application. For example, a key-value store might wish to tag a record that has Put information with the key that is written. For this reason, we utilize distributed Indexers that provide access to the Log maintainers by tag information. Distributed indexing for distributed shared logs is tackled by several systems [11, 33, 35]

Tag and lookup model. The tag is a string that describes a feature of the record. It is possible that the tag also has a value. Each record might have more than one tag. The application client can lookup a tag by its name and specify the amount of records to be returned. For example, the Application client might lookup records that has a certain tag and request returning the most recent 100 record LIds to be returned with that tag. If the tag has a value attached to it, then the Application client might lookup records with that tag and rules on the value, *e.g.*, look up records with a certain tag with values greater than i and return the most recent x records.

5.4 Challenges

Log gaps. A Log maintainer receiving more records advances in the log ahead of others. For example, Log maintainer A can have 1000 records ready while Log maintainer B has 900 records. This causes temporary gaps in the logs that can be observed by Application clients reading the log. The requirement that needs to be enforced is that *Application clients must not be allowed to read a record at log position i if there exist at least one gap at log position j less than i.*

To overcome the problem of these temporary gaps, minimal gossip is propagated between maintainers. The goal of this gossip is to identify the record LId that will guarantee that any record with a smaller LId can be read from the Log maintainers. We call this LId the *Head of the Log (HL)*. Each Log maintainer has a vector with a size equal to the number of maintainers. Each element in the vector corresponds to the maximum LId at that maintainer. Initially the vector is initialized to all zeros. Each maintainer updates its value in the local vector. Occasionally, a maintainer propagates its maximum LId to other maintainers. When the gossip message is received by a maintainer it updates the corresponding entry in the vector. A maintainer can decide that the HL value is equal to the vector entry with the smallest value. When an application wants to read or know the HL, it asks one of the maintainers for this value. This technique does not pose a significant bottleneck for throughput. This is because it is a fixed-sized gossip that is not dependent on the actual throughput of the shared log. It might, however, cause the latency to be higher as the the throughput increases. This is because of the time required to receive gossip messages and determine whether a LId has no prior gaps.

Explicit order requests. Appends translate to a total order at the datacenter after they are added by the Log maintainers. Concurrent appends therefore do not have precedence relative to each other. It is, however, possible to enforce order for concurrent appends if they were requested by the Application client. One way is to send the appends to the same maintainer in the order wanted. Maintainers ensure that a latter append will have a LId higher than ones received earlier. Otherwise, it is possible to enforce order for concurrent appends across maintainers. The Application client waits for the earlier append to be assigned a LId and then attach this LId as a minimum bound. The maintainer that receives the record with the minimum bound ensures that the record is buffered until it can be added to a partial log with LIds larger than the minimum bound. This solution however must be pursued with care to avoid a large backlog of partial logs.

6. CHARIOTS: GEO-REPLICATED LOG

In this section we show the design of Chariots that supports multi-datacenter replication of the shared log. The design is a multi-stage pipeline that includes FLStore as one of the stages. We begin the discussion by outlining an abstract design of log replication. This abstract design specifies the requirements, guarantees, and interface desired to be provided by Chariots. The abstract solution will act as a guideline in building Chariots, that will be proposed after the abstract design. Chariots is a distributed scale-out platform to manage log replication with the same guarantees and requirements of the abstract solution.

6.1 Abstract solution

Before getting to the distributed solution, it is necessary to start with an efficient **abstract solution**. This abstract solution will be provided here in the form of algorithms running on a totally ordered thread of control at the datacenter. This is similar to saying that the datacenter is the machine and it is manipulating the log according to incoming events. Using this abstract solution, we will design the distributed implementation next (Section 6.2) that will result in a behavior identical to the abstract solution with a higher performance.

The data structures used are a log and a $n \times n$ table, where n is the number of datacenters, called the Awareness Table (ATable) inspired by Replicated Dictionary [36]. The table represents the datacenter's (DC's) extent of knowledge about other DCs. Each row or column represents one DC. Consider DC A and its ATable T_A. The entry $T_A[B,C]$ contains a TOId, τ, that represents B's knowledge about C's records according to A. This means that A is certain that B knows about all records generated at host DC C up to record τ. When a record is added to the log, it is tagged by the following information: (1) *TOId*, (2) *Host datacenter Id*, and (3) causality information.

The body of the record, which is supplied by the application is opaque to Chariots . To do the actual replication, the local log and ATable are continuously being propagated to other DCs. When the log and ATable are received by another DC, the new records are incorporated at the receiving log and the ATable is updated accordingly.

The algorithms to handle operations and propagation are presented with the assumption that only one node manipulates Chariots, containing the log of records and ATable. In Section 6.2 we will build the distributed system that will be manipulating Chariots while achieving the correct behavior of the abstract solution's algorithms presented here. The following are the events that need to be handled by Chariots:

1. **Initialization**: The log is initialized and the ATable entries are set to zero. Note that the first record of each node has a TOId of 1.

2. **Append**: Construct the record by adding the following information: host identifier, TOId, LId, causality, and tags. Update the entry $T_I[I,I]$, where I is the host datacenter's id, to be equal to the record's TOId. Finally, add the record to the log.

3. **Read**: Get the record with the specified LId.

4. **Propagate**: A snapshot of Chariots is sent to another DC j. The snapshot includes a subset of the records in the log that are not already known by j. Whether a record, r, is known to j can be verified using $T_i[j,i]$ and comparing it to $TOId(r)$.

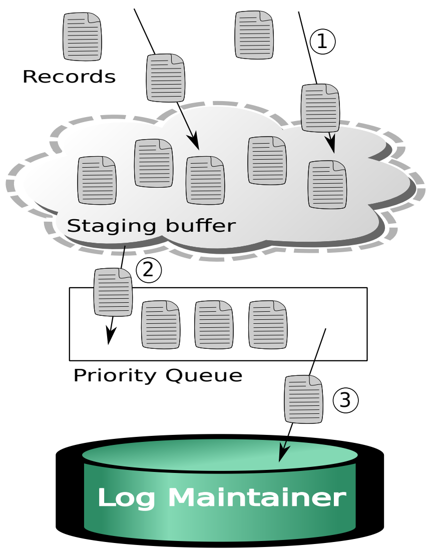


Figure 5: The components involved in adding records in the abstract solution.

5. **Reception**: When a log is received, incorporate all the records that were not seen before to the local log if its causal dependencies are satisfied. Otherwise, add the record with unsatisfied dependencies to a priority queue ordered according to causal relations. This is depicted in Figure 5. The incoming records are all put in a staging buffer (step 1) and are taken and added to the log or priority queue according to their causal dependencies (step 2). Chariots checks the priority queue frequently to transfer any records that have their dependencies satisfied to the log (step 3). Also, the ATable is updated to reflect the newly incorporated records.

Garbage collection. The user has the choice to either garbage collect log records or maintain them indefinitely. Depending on the applications, keeping the log can have great value. If the user choses not to garbage collect the records then they may employ a cold storage solution to archive older records. On the other hand, the user can choose to enable garbage collection of records. It is typical to have a temporal or spatial rule for garbage collecting the log. However, in addition to any rule set by the system designer, garbage collection is performed for records only after they are known by all other replicas. This is equivalent to saying that a record, r, can be garbage collected at i if and only if $\forall_{j \in nodes}(T_i[j, host(r)] \geq ts(r))$, where $host(r)$ is the host node of r.

6.2 Chariots distributed design

In the previous section we showed an efficient abstract design for a shared log that supports multi-datacenter replication. Chariots is a distributed system that mimics that abstract design. Each datacenter runs an instance of Chariots. The shared logs at different datacenters are replicas. All records exist in all datacenters. The system consists of a multi-stage pipeline. Each stage is responsible of performing certain tasks to incoming records and pushing them along the pipeline where they eventually persist in the shared log. Each stage in Chariots is designed to be elastic. An important design principle is that Chariots is designed to identify bottlenecks in the pipeline and allow overcoming them by adding more resources to the stages that are overwhelmed. For this to be successful, elasticity of each stage is key. Minimum to no dependencies exist between the machines belonging to one stage.

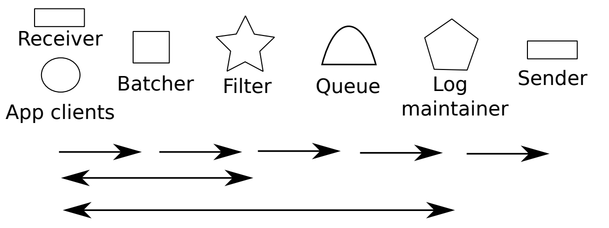


Figure 6: The components of the multi-data center shared log. Arrows denote communication pathways in the pipeline.

Pipeline design. Chariots pipeline consists of six stages depicted in Figure 6. The first stage contains nodes that are generating records. These are Application clients and machines receiving the records sent from other datacenters. These records are sent to the next stage in the pipeline, *Batchers*, to batch records to be sent collectively to the next stage. *Filters* receive the batches and ensure the uniqueness of records. Records are then forwarded to Queues where they are assigned LId. After assigning a LId to a record it is forwarded to FLStore that constitutes the Log maintainers stage. The local records in the log are read from FLStore and sent to other datacenters via the *Senders*.

The arrows in Figure 6 represent the flow of records. Generally, records are passed from one stage to the next. However, there is an exception. Application clients can request to read records from the Log maintainers. Chariots support elastic expansion of each stage to accommodate increasing demand. Thus, each stage can consist of more than one machine, *e.g.*, five machines acting as Queues and four acting as Batchers. The following is a description of each stage:

Application clients. The Application client hosts the application modules. These modules uses the interface to the log that was presented in Section 3. Some of the commands are served by only reading the log. These include Read and control commands. These requests are sent directly to the Log maintainers. The Append operation creates a record that encapsulates the user's data and send it to any Batcher machine.

Batchers. The Batchers buffer records that are received locally or from external sources. Batchers are completely independent from each other, meaning that no communication is needed from one Batcher to another and that scaling to more batchers will have no overhead. Each Batcher has a number of buffers equal to the number of Filters. Each record is mapped to a specific Filter to be sent to it eventually. Once a buffer size exceeds a threshold, the records are sent to the designated Filter. The way records are mapped to Filters is shown next.

Filters. The Filters ensures uniqueness of records. To perform this task, each Filter becomes a champion for a subset of the records. One natural way to do so is to make each Filter a champion for records with the same host Id, i.e. the records that were created at the same datacenter. If the number of Filters needed is less that the number of datacenters, then a single Filter can be responsible for more than one datacenter. Otherwise, if the number of needed Filters is in fact larger than the number of datacenters, then more than one Filter need to be responsible for a single datacenter's records. For example, consider that two Filters, x and y, responsible for records coming from datacenter A. Each one can be responsible for a subset of the records coming from A. This can be achieved by leveraging the unique, monotonically increasing, TOIds. Thus, x can be responsible for ensuring uniqueness of A's records with odd

TOIds and *y* can ensure the uniqueness of records with even TOIds. Of course, any suitable mapping can be used for this purpose. To ensure uniqueness, the processing agent maintains a counter of the next expected TOId. When the next expected record arrives it is added to the batch to be sent to the one of the Queues. Note also that this stage does not require any communication between filters, thus allowing seamless scalability.

Queues. Queues are responsible for assigning LIds to the records. This assignment must preserve the ordering guarantees of records. To append records to the shared log they need to have all their causal dependencies satisfied in addition to the total order of records coming from the same datacenter. Once a group of records have their causal dependencies satisfied, they are assigned LIds and sent to the appropriate log maintainer for persistence. For multi-datacenter records with causal dependencies, it is not possible to append to the FLStore directly and make it assign LIds in the same manner as the single-datacenter deployment shown in section 5.2. This is because it is not guaranteed that any record can be added to the log at any point in time, rather, its dependencies must be satisfied first. The queues ensure that these dependencies are preserved and assign LIds for the records before they are sent to the log maintainers. The queues are aware of the deterministic assignment of LIds in the log maintainers and forward the records to the appropriate maintainer accordingly.

Queues ensure causality of LId assignments by the use of a token. The token consists of the current maximum TOId of each datacenter in the local log, the LId of the most recent record, and the deferred records with unsatisfied dependencies. The token is initially placed at one of the Queues. The Queue holding the token append all the records that can be added to the log. The Queue can verify whether a record can be added to the shared log by examining the maximum TOIds in the token. The records that can be added are assigned LIds and sent to the Maintainers designated for them. The token is updated to reflect the records appended in the log. Then, the token is sent to the next maintainer in a round-robin fashion. The token might include all, some, or none of the records that were not successfully added to the log. Including more deferred records with the token consumes more network I/O. On the other hand, not forwarding the deferred records with the token might increase the latency of appends. It is a design decision that depends on the nature of Chariots deployment.

Log maintainers. These Log maintainers are identical to the distributed shared log maintainers of FLStore presented in Section 5.2. Maintainers ensure the persistence of records in the shared log. The record is available to be read by senders and application clients when they are persisted in the maintainers.

Log propagation. Senders propagate the local records of the log to other datacenters. Each sender is limited by the I/O bandwidth of its network interface. To enable higher throughputs, more Senders are needed at each datacenter. Likewise, more Receivers are needed to receive the amount of records sent. Each Sender machine is responsible to send parts of the log from some of the maintainers to a number of Receivers at other datacenters.

6.3 Live elasticity

The demand on web and cloud applications vary from time to time. The ability of the infrastructure to scale to the demand seamlessly is a key feature for its success. Here, we show how adding compute resources to Chariots in the fly is possible without disruptions to the Application clients. The elasticity model of Chariots is to treat each stage as an independent unit. This means that it is possible to add resources to a single stage to increase its capacity without affecting the operation of other stages.

Completely independent stages. Increasing the capacity of completely independent stages merely involves adding the new resources and sending the information of the new machine to the higher layer. The completely independent stages are the receivers, batchers, and senders. For adding a receiver, the administrator needs to inform senders of other datacenters so that it can be utilized. Similarly, a new batcher need to inform local receivers of its existence. A new sender is different in that it is the one reading from log maintainers, thus, the log maintainers need not be explicitly told about the introduction of a new sender.

Filters. In Chariots, each filter is championing a specific subset of the log records. Increasing the number of filters results in the need of reassigning championing roles. For example, a filter that was originally championing records from another datacenter could turn out to be responsible for only a subset of these records while handing off the responsibility of the rest of them to the new filter. This reassignment need to be orchestrated with batchers. There need to be a way for batchers to figure out when the hand-over took place so that they can direct their records accordingly. A *future reassignment* technique will be followed for filters as well as log maintainers as we show next. A future reassignment for filters begin by marking future TOIds that are championed by the original filter. These future TOIds mark transition of championing a subset of the records to the new filter. Consider a filter that champions records from datacenter *A* in a reassignment scenario of adding a new filter that will champion the subset of these records with even TOIds. Marking a future TOId, *t*, will result in records with even TOIds greater than *t* to be championed by the new filter. This future reassignment should allow enough time to propagate this information to batchers.

Queues. Adding a new queue involves two tasks: making the new queue part of the token exchange loop and propagating the information of its addition to filters. The first task is performed by informing one of the queues that it should forward the token to the new queue rather than the original neighbor. The latter task (informing filters) can be performed without coordination because a queue can receive any record.

Log maintainers. Expanding log maintainers is similar to expanding filters in that each maintainer champions a specific set of records. In this case, each log maintainer champions a subset of records with specific LIds. The future reassignment technique is used in a similar way to expanding filters. In this case, not only do the queues need to know about the reassignment, but the readers need to know about it too. Another issue is that log maintainers persist old records. Rather than migrating the old records to the new champion, it is possible to maintain an epoch journal that denotes the changes in log maintainer assignments. These can be used by readers to figure out which log maintainer to ask for an old record.

7. EVALUATION

In this section we present some experiments to evaluate our implementation of FLStore and Chariots. The experiments were conducted on a cluster with nodes with the following specifications. Intel Xeon E5620 CPUs that are running 64-bit CentOS Linux with OpenJDK 1.7 were used. The nodes in a single rack are connected by a 10GB switch with an average RTT of 0.15 ms. We also perform the baseline experiments on Amazon AWS. There, we used compute optimized machines (c3.large) in the datacenter in Virginia. Each machine has 2 virtual CPUs and a 3.75 GiB memory. We refer to the earlier setup as the *private cloud* and the latter setup as the *public cloud*. Unless it is mentioned otherwise, the size of each record is 512 Bytes.

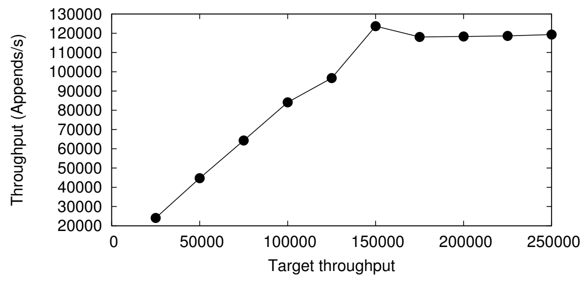


Figure 7: The throughput or one maintainer while increasing the load in a public cloud

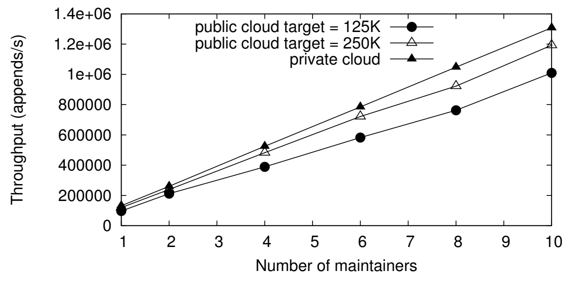


Figure 8: The append throughput of the shared log in a single-datacenter deployment while increasing the number of Log Maintainers.

7.1 FLStore scalability

The first set of experiments that we will present is of the FLStore implementation which operates within the datacenter. Each one of the experiments consists of two types of machines, namely Log maintainers and clients. The clients generate records and send them to the Log Maintainers to be appended. We are interested in measuring the scaling behavior of FLStore while increasing the number of maintainers. We begin by getting a sense of the capacity of the machines. Figure 8 shows the throughput of one maintainer in the public cloud while increasing the load on it. Records are generated with a specific rate at each experiment point from other machines. The rate is called the *target throughput*. Note how as the target throughput increases, the achieved throughput increases up to a point and then plateaus. The maximum throughput is achieved when the target throughput is 150K and then drops to be around 120K appends per second. These numbers will help us decide what target throughputs to choose for our next experiments.

To verify the scalability of FLStore, Figure 8 shows the cumulative throughput of different scenarios each with a different number of maintainers. For each experiment an identical number of client machines were used to generate records to be appended. Ideally, we would like the throughput to increase linearly with the addition of new maintainers. Three plots are shown, two from the public cloud and one from the private cloud. The ones from the public cloud differ in the target throughput to each maintainer. One targets 125K appends per second for each maintainer while the other targets 250K appends per second. Note how one is below the plateau point and one is above. The figure shows that FLStore scales with the addition of resources. A single maintainer has a throughput of 131K for the private cloud, 96.7K for the public cloud with a target of 125K, and 119K for the public cloud with the target of 250K. As we are increasing the number of Log Maintainers a near-linear scaling is observed. For ten Log Maintainers, the achieved append throughput was 1308034 record appends per second for the private cloud. This append throughput is 99.3% when compared to a perfect scaling case. The public cloud case with a target of 125K achieves a throughput that is slightly larger than the perfect scaling case. This is due to the variation in machines' performances. The other public cloud case achieve a scaling of 99.9%. This near-perfect scaling of FLStore is encouraging and demonstrates the effect of removing any dependencies between maintainers.

Machine	Throughput (Kappends/s)
Client	129
Batcher	129
Filter	129
Maintainer	124
Store	132

Table 2: The throughput of machines in a basic deployment of Chariots with one machine per stage.

Machine	Throughput (Kappends/s)
Client 1	120
Client 2	122
Batcher	126
Filter	125
Maintainer	123
Store	132

Table 3: The throughput of machines in a deployment of Chariots with two clients and one machine per stage for the remaining stages.

7.2 Chariots scalability

The full deployment of Chariots that is necessary to operate in a multi-datacenter environment consists of five stages. These stages are described in Section 6.2. Here, we will start from a basic deployment of one machine per stage in the private cloud. We observe the throughput of each stage to try to identify the bottleneck. Afterward, we observe how this bottleneck can be overcome by increasing resources. The simple deployment of one machine per stage of Chariots pipeline achieves the throughputs shown in Table 2. The table lists the throughput in Kilo records per second for each machine in the pipeline. Note how all machines achieve a similar throughput of records per second. It is possible for the store to achieve a throughput higher than the client because of the effect of buffering. Close throughput numbers for all machines indicates that the bottleneck is possibly due to the clients. The clients might be generating less records per second than what can be handled by the pipeline.

To test this hypothesis we increase the number of machines generating records to two client machines. The results are shown in Table 3. If the clients were indeed not generating enough records to saturate the pipeline, then we should observe an increase in the throughput of the Batcher. However, this was not the case. The increased load actually resulted in a lower throughput for the batcher. This means that the batcher is possibly the bottleneck. So, we increase the number of batchers to observe the throughput of latter stages in the pipeline. Table 4 shows the throughput of machines with two client machines, two batchers, and a single machine for each of the remaining stages. Both batchers achieve a throughput that is higher than the one achieved by a single batcher in the pre-

Machine	Throughput (Kappends/s)
Client 1	126
Client 2	129
Batcher 1	149
Batcher 2	129
Filter	120
Maintainer	118
Store	121

Table 4: The throughput of machines in a deployment of Chariots with two client machines, two Batchers, and a single machine for the remaining stages.

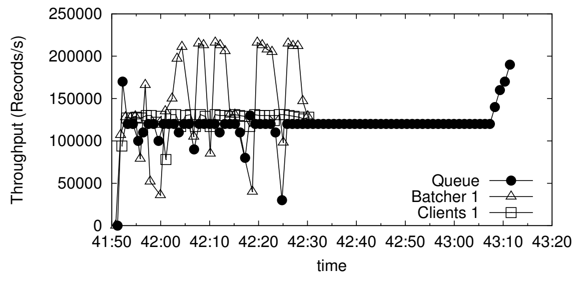


Figure 9: The throughput of machines in a deployment of Chariots with two client machines, two Batchers, and a single machine for the remaining stages

vious experiments. This means that the throughput of the Batcher stage more than doubled. However, now the bottleneck is pushed to the filter stage that is not able to handle more than 130000 records per second. Because the throughput of latter stages is almost half the throughput of the Batcher, they take twice the time to finish the amount of records generated by the clients (10000000 records). The throughput timeseries for one client, one batcher, and the queue are shown in Figure 9. We did not show all the machines' throughputs to avoid cluttering the figure. The Batchers are done with the records at time 42:30, whereas, the latter stages lasted till time 43:10. Note that by the end of the experiment, the throughput of the queue increases abruptly. The reason for this increase is that the although the Batchers had already processed the records they are still transmitting them to the Filter until time 43:08, right before the abrupt increase. The network interface's I/O of the Filter was limiting its throughput. After it is no longer receiving from the two Batchers it can send with a higher capacity to the latter stages, thus causing an increase in the observed throughput. This is also the reason of why in the beginning of the experiment, a higher throughput is observed for some of the stages (*e.g.*, the high throughput in the beginning for the queue). The reason is that they still had capacity in their network's interface I/O before it was also used to propagate records to latter stages. Another interesting observation is the performance variation of the batcher. This turned out to be a characteristic of machines at a stage generating more throughput than what can be handled by the next stage.

Increasing the number of machines further should yield a better throughput. We experiment with the previous setting, but this time with two machines for all stages. The throughput values records are presented in Table 5. Note how all stages are scaling. The throughput of each stage has doubled. Each machine achieves a close throughput to the basic case of a pipeline with one machine per stage.

Machine	Throughput (Kappends/s)
Client 1	130
Client 2	130
Batcher 1	127
Batcher 2	127
Filter 1	127
Filter 2	126
Maintainer 1	125
Maintainer 2	126
Store 1	137
Store 2	137

Table 5: The throughput of machines in a deployment of Chariots with two machines per stage.

Our main objective is to allow scaling of shared log systems to support today's applications. We showed in this evaluation how a FLStore deployment is able to scale while increasing the number of maintainers within the datacenter (Figure 8). Also, we evaluated the full Chariots pipeline that is designed to be used for multi-datacenter environments. The bottleneck of a basic deployment was identified and Chariots overcome it by adding more resources to the pipeline.

8. CONCLUSION

In this paper we presented a shared log system called Chariots. The main contribution of Chariots is the design of a distributed shared log system that is able to scale beyond the limit of a single node. This is enabled by a deterministic post-assignment approach of assigning ranges of records to Log maintainers. Chariots also increases the level of availability and fault-tolerance by supporting geo-replication. A novel design to support a shared log across datacenters is presented. Causal order is maintained across records from different datacenters. To allow scaling such a shared log, a multi-stage pipeline is proposed. Each stage of the pipeline is designed to be scalable by minimizing the dependencies between different machines. An experimental evaluation demonstrated that Chariots is able to scale with increasing demand by adding more resources.

9. ACKNOWLEDGMENTS

This work is supported by NSF grant IIS 1018637 and is partially funded by a gift grant from NEC Labs America. Faisal Nawab is partially funded by a scholarship from King Fahd University of Petroleum and Minerals. We would also like to thank Amaozn for access to Amazon EC2.

References

[1] J. Bonér. The Road to Akka Cluster and Beyond. https://www.youtube.com/watch?v=2wSYcyWCtx4.

[2] N. Marz. How to beat the CAP theorem. http://bit.ly/marz-cap-theorem.

[3] P. Helland. Immutability changes everything! http://vimeo.com/52831373. Talk at RICON, 2012.

[4] R. Ananthanarayanan et al. Photon: Fault-tolerant and scalable joining of continuous data streams. In *SIGMOD*, pages 577–588. ACM, 2013.

[5] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on causal consistency. In *SIGMOD*, pages 761–772. ACM, 2013.

[6] J. Baker, C. Bond, J. Corbett, J. J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, pages 223–234, 2011.

[7] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. D. Davis. Corfu: A shared log design for flash clusters. In *NSDI*, pages 1–14, 2012.

[8] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. Tango: Distributed data structures over a shared log. In *SOSP*, pages 325–340. ACM, 2013.

[9] L. A. Barroso and U. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 4(1):1–108, 2009.

[10] N. M. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. Practi replication. In *NSDI*, 2006.

[11] P. Bernstein, C. Reid, and S. Das. Hyder-a transactional record manager for shared flash. In *CIDR*, pages 9–20, 2011.

[12] E. A. Brewer. Towards robust distributed systems. In *PODC*, page 7, 2000.

[13] N. Conway, P. Alvaro, E. Andrews, and J. M. Hellerstein. Edelweiss: Automatic storage reclamation for distributed programming. *Proceedings of the VLDB Endowment*, 7(6), 2014.

[14] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, Aug. 2008.

[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[16] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[17] A. Gupta, F. Yang, J. Govig, A. Kirsch, K. Chan, K. Lai, S. Wu, S. G. Dhoot, A. R. Kumar, A. Agiwal, et al. Mesa: Geo-replicated, near real-time, scalable data warehousing. *Proceedings of the VLDB Endowment*, 7(12), 2014.

[18] J. H. Hartman and J. K. Ousterhout. The zebra striped network file system. *ACM Transactions on Computer Systems (TOCS)*, 13(3):274–310, 1995.

[19] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems (TOCS)*, 10(4):360–391, 1992.

[20] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[21] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[22] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[23] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In *SOSP*, pages 401–416. ACM, 2011.

[24] P. Mahajan, L. Alvisi, and M. Dahlin. Consistency, availability, and convergence. *University of Texas at Austin Tech Report*, 11, 2011.

[25] J. Mickens, E. B. Nightingale, J. Elson, K. Nareddy, D. Gehring, B. Fan, A. Kadav, V. Chidambaram, and O. Khan. Blizzard: fast, cloud-scale block storage for cloud-oblivious applications. In *NSDI*, pages 257–273. USENIX, 2014.

[26] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. *ACM SIGOPS Operating Systems Review*, 36(SI):31–44, 2002.

[27] F. Nawab, D. Agrawal, and A. El Abbadi. Message futures: Fast commitment of transactions in multi-datacenter environments. In *CIDR*, 2013.

[28] F. Nawab, V. Arora, D. Agrawal, and A. E. Abbadi. Minimizing commit latency of transactions in geo-replicated data stores. In *SIGMOD*, 2015.

[29] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in ramcloud. In *SOSP*, pages 29–41. ACM, 2011.

[30] S. Patterson, A. J. Elmore, F. Nawab, D. Agrawal, and A. E. Abbadi. Serializability, not serial: Concurrency control and availability in multi-datacenter datastores. *PVLDB*, 5(11):1459–1470, 2012.

[31] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. *Flexible update propagation for weakly consistent replication*, volume 31. ACM, 1997.

[32] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *SOSP*. ACM, 1995.

[33] H. T. Vo, S. Wang, D. Agrawal, G. Chen, and B. C. Ooi. Logbase: a scalable log-structured database system in the cloud. *Proceedings of the VLDB Endowment*, 5(10):1004–1015, 2012.

[34] M. Vrable, S. Savage, and G. M. Voelker. Bluesky: a cloud-backed file system for the enterprise. In *FAST*, page 19, 2012.

[35] S. Wang, D. Maier, and B. C. Ooi. Lightweight indexing of observational data in log-structured storage. *Proceedings of the VLDB Endowment*, 7(7), 2014.

[36] G. T. Wuu and A. J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, PODC '84, pages 233–242, New York, NY, USA, 1984. ACM.

Hermes: Dynamic Partitioning for Distributed Social Network Graph Databases

Daniel Nicoara
University of Waterloo
daniel.nicoara@gmail.com

Shahin Kamali
University of Waterloo
s3kamali@uwaterloo.ca

Khuzaima Daudjee
University of Waterloo
kdaudjee@uwaterloo.ca

Lei Chen
HKUST
leichen@cse.ust.hk

ABSTRACT

Social networks are large graphs that require multiple graph database servers to store and manage them. Each database server hosts a graph partition with the objectives of balancing server loads, reducing remote traversals (edge-cuts), and adapting the partitioning to changes in the structure of the graph in the face of changing workloads. To achieve these objectives, a dynamic repartitioning algorithm is required to modify an existing partitioning to maintain good quality partitions while not imposing a significant overhead to the system. In this paper, we introduce a *lightweight repartitioner*, which dynamically modifies a partitioning using a small amount of resources. In contrast to the existing repartitioning algorithms, our lightweight repartitioner is efficient, making it suitable for use in a real system. We integrated our lightweight repartitioner into Hermes, which we designed as an extension of the open source Neo4j graph database system, to support workloads over partitioned graph data distributed over multiple servers. Using real-world social network data, we show that Hermes leverages the lightweight repartitioner to maintain high quality partitions and provides a 2 to 3 times performance improvement over the de-facto standard random hash-based partitioning.

1. INTRODUCTION

Large scale graphs, in particular social networks, permeate our lives. The scale of these networks, often in millions of vertices or more, means that it is often infeasible to store, query and manage them on a single graph database server. Thus, there is a need to partition, or shard, the graph across multiple database servers, allowing the load and concurrent processing to be distributed over these servers to provide good performance and increase availability. Social networks exhibit a high degree of correlation for accesses of certain groups of records, for example through frictionless sharing [15]. Also, these networks have a heavy-tailed distribution for popularity of vertices. To achieve a good partitioning which improves the overall performance, the following objectives need to be met:

- The partitioning should be *balanced*. Each vertex of the graph has a *weight* that indicates the popularity of the vertex (e.g., in terms of the frequency of queries to that vertex). In social networks, a small number of users (e.g., celebrities, politicians) are extremely popular while a large number of users are much less popular. This discrepancy reveals the importance of achieving a balanced partitioning in which all partitions have almost equal *aggregate weight* defined as the total weight of vertices in the partition.
- The partitioning should minimize the number of *edge-cuts*. An edge-cut is defined by an edge connecting vertices in two different partitions and involves queries that need to transition from a partition on one server to a partition on another server. This results in shifting local traversal to remote traversal, thereby incurring significant network latency. In social networks, it is critical to minimize edge-cuts since most operations are done on the node that represents a user and its immediate neighbors. Since these *1-hop traversal* operations are so prevalent in these networks, minimizing edge-cuts is analogous to keeping communities intact. This leads to highly local queries similar to those in SPAR [27] and minimizes the network load, allowing for better scalability by reducing network IO.
- The partitioning should be *incremental*. Social networks are *dynamic* in the sense that users and their relations are always changing, e.g., a new user might be added, two users might get connected, or an ordinary user might become popular. Although the changes in the social graph can be much slower when compared to the read traffic [8], a good partitioning solution should dynamically adapt its partitioning to these changes. Considering the size of the graph, it is infeasible to create a partitioning from scratch; hence, a repartitioning solution, a *repartitioner*, is needed to improve on an existing partitioning. This usually involves *migrating* some vertices from one partition to another.
- The repartitioning algorithm should perform well in terms of time and memory requirements. To achieve this efficiency, it is desirable to perform repartitioning locally by accessing a small amount of information about the structure of the graph. From a practical point of view, this requirement is critical and prevents us from applying existing approaches, e.g., [18, 30, 31, 6] for the repartitioning problem.

The focus of this paper is on the design and provision of a *practical* partitioned social graph data management system that can support remote traversals while providing an

10.5441/002/edbt.2015.04

effective method to *dynamically* repartition the graph using only local views. The distributed partitioning aims to co-locate vertices of the graph *on-the-fly* so as to satisfy the above requirements. The fundamental contribution of this paper is a dynamic partitioning algorithm, referred to as *lightweight repartitioner*, that can identify which parts of graph data can benefit from co-location. The algorithm aims to incrementally improve an existing partitioning by decreasing edge-cuts while maintaining almost balanced partitions. The main advantage of the algorithm is that it relies on only a small amount of knowledge on the graph structure referred to as *auxiliary data*. Since the auxiliary data is small and easy to update, our repartitioning algorithm is performant in terms of time and memory while maintaining high-quality partitionings in terms of edge-cut and load balance.

We built Hermes as an extension of the Neo4j[1] open source graph database system by incorporating into it our algorithm to provide the functionality to move data on-the-fly to achieve data locality and reduce the cost of remote traversals for graph data. Our experimental evaluation of Hermes using real-world social network graphs shows that our techniques are effective in producing performance gains and work almost as well as the popular Metis partitioning algorithms [18, 30, 6] that performs static offline partitioning by relying on a global view of the graph.

The rest of the paper is structured as follows. Section 2 describes the problem addressed in the paper and reviews classical approaches and their shortcomings. Section 3 introduces and analyzes the lightweight repartitioner. Section 4 presents an overview of the Hermes system. Section 5 presents performance evaluation of the system. Section 6 covers related work, and Section 7 concludes the paper.

2. PROBLEM DEFINITION

In this section we formally define the partitioning problem and review some of the related results. In what follows, the term 'graph' refers to an undirected graph with weights on vertices.

2.1 Graph Partitioning

In the classical (α, γ)-graph partitioning problem [20], the goal is to partition a given graph into α vertex-disjoint subgraphs. The weight of a partition is the total weight of vertices in that partition. In a *valid solution*, the weight of each partition is at most a factor $\gamma \geq 1$ away from the average weight of partitions. More precisely, for a partition P of a graph G, we need to have $\omega(P) \leq \gamma \times \sum_{v \in V(G)} \omega(v)/\alpha$. Here, $\omega(P)$ and $\omega(v)$ denote the weight of a partition P and vertex v, respectively. Parameter γ is called the *imbalance load factor* and defines how imbalanced the partitions are allowed to be. Practically, γ is in range $[1, 2]$. Here, $\gamma = 1$ implies that partitions are required to be completely balanced (all have the same aggregate weights), while $\gamma = 2$ allows the weight of one partition to be up to twice the average weight of all partitions. The goal of the minimization problem is to achieve a valid solution in which the number of edge-cuts is minimized.

The partitioning problem is NP-hard [13]. Moreover, there is no approximation algorithm with a constant approximation ratio unless P=NP [7]. Hence, it is not possible to introduce algorithms which provide worst-case guarantees on the quality of solutions, and it makes more sense to study the typical behavior of algorithms. Consequently, the problem is mostly approached through heuristics [20] [12] which are aimed to improve the average-case performance. Regardless, the time complexity of these heuristics $\Omega(n^3)$ which makes them unsuitable in practice.

To improve the time complexity, a class of *multi-level* algorithms were introduced. In each *level* of these algorithms, the input graph is *coarsened* to a representative graph of smaller size; when the representative graph is small enough, a partitioning algorithm like that of Kernighan-Lin [20] is applied to it, and the resulting partitions are mapped back (uncoarsened) to the original graph. Many algorithms fit in this general framework of multi-level algorithms; a widely used example is the family of Metis algorithms [19, 30, 6]. The multi-level algorithms are *global* in the sense that they need to know the whole structure of the graph in the coarsening phase, and the coarsened graph in each stage should be stored for the uncoarsening stage. This problem is partially solved by introducing distributed versions of these algorithms in which the partitioning algorithm is performed in parallel for each partition [4]. In these algorithms, in addition to the local information (structure of the partition), for each vertex, the list of the adjacent vertices in other partitions is required in the coarsening phase. The following theorem establishes that in the worst case, acquiring this amount of data is close to having a global knowledge of graph (the proof can be found in [25]).

THEOREM 1. *Consider the (α, γ)-graph partitioning problem where $\gamma < 2$. There are instances of the problem for which the number of edge-cuts in any valid solution is asymptotically equal to the number of edges in the input graph.*

Hence, the average amount of data required in the coarsening phase of multi-level algorithms can be a constant fraction of all edges. The graphs used in the proof of the above theorem belong to the family of power-law graphs which are often used to model social networks. Consequently, even the distributed versions of multi-level algorithms in the worst case require *almost* global information on the structure of the graph (particularly when used for partitioning social networks). This reveals the importance of providing practical partitioning algorithms which need only a small amount of knowledge about the structure of the graph that can be easily maintained in memory. The lightweight repartitioner introduced in this paper has this property, i.e., it maintains only a small amount of data, referred to as auxiliary data, to perform repartitioning.

2.2 Repartitioning

A variety of partitioning methods can be used to create an initial, *static*, partitioning. This should be followed by a *repartitioning* strategy to maintain good partitioning that can adapt to changes in the graph. One solution is to periodically run an algorithm on the whole graph to get new partitions. However, running an algorithm to get new partitions from scratch is costly in terms of time and space. Hence, an incremental partitioning algorithm needs to adapt the existing partitions to changes in the graph structure.

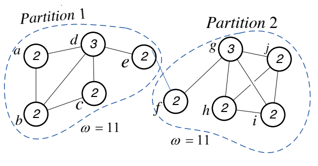
It is desirable to have a lightweight repartitioner that maintains only a small amount of auxiliary data to perform

[1]Neo4j is being used by customers such as Adobe and HP [3].

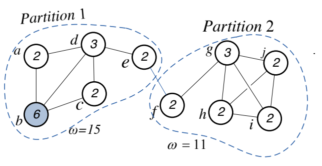
repartitioning. Since such algorithm refers only to this auxiliary data, which is significantly smaller than the actual data required for storing the graph, the repartitioning algorithm is not a system performance bottleneck. The auxiliary data maintained at each machine (partition) consists of the list of accumulated weight of vertices in each partition, as well as the *number* of neighbors of each hosted vertex in each partition. Note that maintaining the number of neighbors is far cheaper that maintaining the *list* of neighbors in other partitions. In what follows, the main ideas behind our lightweight repartitioner are introduced through an example.

Example: Consider the partitioning problem on the graph shown in Figure 1. Assume there are $\alpha = 2$ partitions in the system and the imbalance factor is $\gamma = 1.1$, i.e., in a valid solution, the aggregate weight of a partition is at most 1.1 times more than the average weight of partitions. Assume the numbers on vertices denote their weight. During normal operation in social networks, users will request different pieces of information. In this sense, the weight of a vertex is the number of read requests to that vertex. Figure 1a shows a partitioning of the graph into two partitions, where there is only one edge-cut and the partitions are well balanced, i.e., the weight of both partitions is equal to the average weight. Assuming user b is a popular weblogger who posts a post, the request traffic for vertex b will increase as its neighbors poll for updates, leading to an imbalance in load on the first partition (see Figure 1b). Here, the ratio between aggregate weight of partition 1 (i.e., 15) and the average weight of partitions (i.e., 13) is more than γ. This means that the response time and request rates increase by more than the acceptable skew limit, and the repartitioning needs to be triggered to rebalance the load across partitions (while keeping the number of edge-cuts as small as possible).

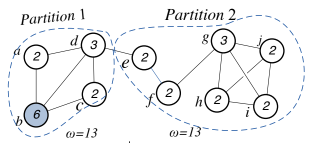
The auxiliary data of the lightweight repartitioner available to each partition includes the weight of each of the two partitions, as well as the number of neighbors of each vertex v hosted in the partition. Provided with this auxiliary data, a partition can determine whether load imbalances exist and the extent of the imbalance in the system (to compare it with γ). If there is a load imbalance, a repartitioner needs to indicate where to migrate data to restore load balance. Migration is an iterative process which will identify vertices that when moved will balance loads (aggregate weights) while keeping the number of edge-cuts as small as possible. For example, when the repartitioner starts from the state in Figure 1b, on partition 1, vertices a through d are poor candidates for migration because their neighbors are in the same partition. Vertex e, however, has a split access pattern between partitions 1 and 2. Since vertex e has the fewest neighbors in partition one, it will be migrated to partition 2. On partition 2, the same process is performed in parallel; however, vertex f will not be migrated since partition 1 has a higher aggregate weight. Once vertex e is migrated, the load (aggregate weights) becomes balanced, thus any remaining iterations will not result in any migrations (see Figure 1c).



(a) Balanced partitioned graph



(b) Skewed graph



(c) Repartitioned graph

Figure 1: Graph evolution and effects of repartitioning in response to imbalances.

The above example is a simple case to illustrate how the lightweight repartitioner works. Several issues are left out of the example, e.g., two highly connected clusters of vertices may repeatedly exchange their clusters to decrease edge-cut. This results in an *oscillation* which is discussed in detail in Section 3.

3. PARTITIONING ALGORITHM

Unlike Neo4j which is centralized, Hermes can apply hash-based or Metis algorithm to partition a graph and distribute the partitions to multiple servers. Thus, the system starts with an initial partitioning and incrementally applies the lightweight repartitioner to maintain partitioning with good performance in the dynamic environment. In this section, we introduce the lightweight repartitioner algorithm behind Hermes. Embedding the initial partitioning algorithm and the lightweight repartitioner into Neo4j required modification of Neo4j components.

To increase query locality and decrease query response times, the initial partitioning needs to be optimized in terms of having almost balanced distributions (valid solutions) with small number of edge-cuts. We use Metis to obtain the initial data partitioning, which is a static, offline, process that is orthogonal to the dynamic, on-the-fly, partitioning that Hermes performs.

3.1 Lightweight Repartitioner

When new nodes join the network or the traffic patterns (weights) of nodes change, the lightweight repartitioner is triggered to rebalance vertex weights while decreasing edge-cut through an iterative process. The algorithm makes use of aggregate vertex weight information as its auxiliary data. Assuming there are α partitions, for each vertex v, the auxiliary data includes α integers indicating the number of neighbors of v in each of the α partitions. This auxiliary data is insignificant compared to the *physical data* associated with the vertex which include adjacency list and other information referred to as *properties* of the vertex (e.g., pictures posted by a user in a social network). The repartitioning auxiliary data is collected and updated based on execution of user requests, e.g., when a new edge is added, the auxiliary data of the partitioning(s) including the endpoints of the edge get updated (two integers are incremented). Hence, the cost involved in maintenance of auxiliary data is proportional to the rate of changes in the graph. As mentioned earlier, social networks change quite slowly (when compared to the read traffic); hence, the maintenance of auxiliary data is not a system bottleneck. Each partition collects and stores aggregate vertex information relevant to only the local vertices. Moreover, the auxiliary data includes the total weight of all partitions, i.e., in doing repartitioning, each server knows the total weight of all other partitions.

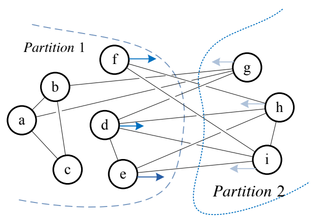
The repartitioning process has two *phases*. In each iteration of the first phase, each server runs the repartitioner algorithm using the auxiliary data to indicate some vertices in its partition that should be migrated to other partitions. Before the next iteration, these vertices are *logically* moved to their target partitions. Logical movement of a vertex means that only the auxiliary data associated with the vertex is sent to the other partition. This process continues up to a point (iteration) in which no further vertices are chosen for migration. At this point the second phase is performed in which the physical data is moved based on the result of first phase. The algorithm is split into two phases because border vertices are likely to change partitions more than once (this will be discussed later) and auxiliary data records are lightweight compared to the physical data records, allowing the algorithm to finish faster. In what follows, we describe how vertices are selected for migration in an iteration of the repartitioner.

Consider a partition P_s (source partition) is running the repartitioner algorithm. Let v be a vertex in partition P_s. The *gain* of moving v from P_s to another partition P_t (target partition) is defined as the difference between the number of neighbors of v in P_t and P_s, respectively, i.e., $gain(v) = d_v(t) - d_v(s)$ ($d_v(k)$ denotes the number of neighbors of v in partition k). Intuitively, the gain represents the decrease of the number of edge-cuts when migrating v from P_s to P_t (assuming that no other vertex migrates). Note that the gain can be negative, meaning that it is better, in terms of edge-cuts, to keep v in P_s rather than moving it to P_t. In each iteration and on each partition, the repartitioner selects for migration *candidate* vertices that will give the maximum gain when moved from the partition. However, to avoid *oscillation* and ensure a valid packing in term of load balance, the algorithm enforces a set of rules in migrating vertices. First, it defines two *stages* in each iteration. In the first stage, the migration of vertices is allowed only from partitions with lower ID to higher ID, while the second stage allows the migration only in the opposite direction, i.e., from partitions with higher ID to those with lower ID. Here, partition ID defines a fixed ordering of partitions (and can be replaced by any other fixed ordering). Migrating vertices in one-direction in two stages prevent the algorithm from oscillation. Oscillation happens when there is a large number of edges between two group of vertices hosted in two different partitions (see Figure 2). If the algorithm allows two-way migration of vertices, the vertices in each group migrate to the partition of the other group, while the edge-cut does not improve (Figure 2b). In one-way migration, however, the vertices in one group remain in their partitions while the other group joins them in that partition (Figure 2d).

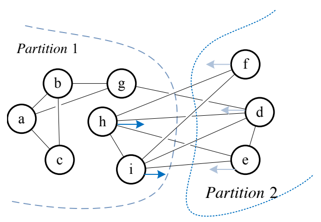
In addition to preventing oscillation, the repartitioner algorithm minimizes load imbalance as follows. A vertex v on a partition P_s is a candidate for migration to partition P_t if the following conditions hold:

- P_s and P_t fulfill the above one-way migration rule.
- Moving v from P_s to P_t does not cause P_t to be *overloaded* nor P_s to be *underloaded*. Recall from Section 2.1 that the imbalance ratio of a partition is the ratio between the weight of the partition (the total weight of vertices it is hosting) and the average weight of all the partitions. A partition is overloaded if its imbalance load is more than γ and underloaded if its weight is less than $2 - \gamma$ times the average partition weight. Here, γ is the maximum allowed imbalance factor ($1 < \gamma < 2$); the default value of γ in Hermes is set to be 1.1, i.e., a partition's load is required to be in range $(0.9, 1.1)$ of the average partition weight. This is so that imbalances do not get too high before repartitioning triggers.
- Either P_s is overloaded *OR* there is a positive *gain* in moving v from P_s to P_t. When a partition is overloaded, it is good to consider all vertices as candidates for migration to any other partition as long as they do not cause an overload on the target partition. When the partition is not overloaded, it is good to move only vertices which have positive weight so as to improve the edge-cut.

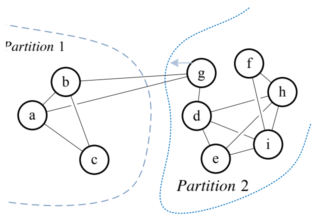
When a vertex v is a candidate for migration to more than one partition, the partition with maximum gain is selected as the target partition of the vertex. This is illustrated in Algorithm 1. Note that detecting whether a vertex v is a candidate for migration and selecting its target partition is performed using only the auxiliary data. Precisely, for detecting underloaded and overloaded partitions (Lines 2, 5 and 11), the algorithm uses the weight of the vertex and the accumulated weights of all partitions; these are included in the auxiliary data. Similarly, for calculating the gain of moving v from partition P_s to partition P_t (Line 10), it uses the number of neighbors of v in any of the partitions, which



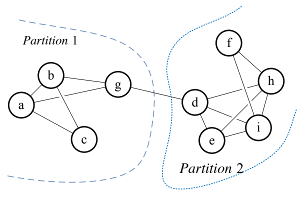
(a) Initial graph, before the first iteration.



(b) The resulted graph if vertices migrate in the same stage (i.e., in a two-way manner).



(c) The resulting graph after the first stage.



(d) The final graph after the second stage.

Figure 2: An unsupervised repartitioning might result in oscillation. Consider the partitioning depicted in (a). The repartitioner on partition 1 detects that migrating d, e, f to partition 2 improves edge-cut; similarly, the repartitioner on partition 2 tends to migrate g, h, i to partition 1. When the vertices move accordingly, as depicted in (b), the edge-cut does not improve and the repartitioner needs to move d, e, f and h, i again. To resolve this issue, in the first stage of repartitioning of (a), the vertices d, e, f are migrated from partition 1 (lower ID) to partition 2 (higher ID). After this, as depicted in (c), the only vertex to migrate in the second stage is vertex g which moves from partition 2 (higher ID) to migration 1 (d).

Algorithm 1 Choosing target partition for migration

```
1: procedure GET_TARGET_PART(vertex $v$ currently hosted in partition $P_s$, the current stage of the iteration.)
2:     if imbalance_factor($P_s - \{v\}$) $< 2 - \gamma$ then
3:         return $(null, 0)$
4:     $target = null$; $maxGain = 0$;
5:     if imbalance_factor($P_s$) $> \gamma$ then
6:         $maxGain = -\infty$
7:     for partition $P_t \in$ partitionSet do
8:         if ($stage = 1$ and $P_t.ID > P_s.ID$) or
9:            ($stage = 2$ and $P_t.ID < P_s.ID$) then
10:            $gain \leftarrow$ Gain($v, P_s, P_t$)
11:            if imbalance_factor($P_t \cup \{v\}$) $< \gamma$ and
12:               $gain > maxGain$ then
13:                $target \leftarrow P_t$; $maxGain = gain$
14:    return $(target, maxGain)$
```

is also included in the auxiliary data.

Recall that the repartitioning algorithm runs on each partition independently, a property that supports scalability. For each partition P_s, after selecting the candidate vertices for migration and their target partitions, the algorithm selects k candidate vertices which have the highest gains among all vertices and proceeds by (logically) migrating these *top-k* vertices to their target partitions. Here, migrating a vertex means sending (and updating) the auxiliary data associated with the vertex to its target destination and updating the auxiliary data associated with partition weights accordingly. The algorithm restricts the number of migrated vertices in each iteration (to k) to avoid imbalanced partitionings. Note that when selecting the target partition for a migrating vertex, the algorithm does not know the target partition of other vertices; hence, there is a chance that a large number of vertices migrate to the same partition to improve edge-cut. Selecting only k vertices enables the algorithm to control the accumulative weight of partitions by restricting the number of migrating vertices. We discuss later how the value of k is selected. In general, taking k as a small, fixed fraction of n (size of the graph) gives satisfactory results.

Algorithm 2 shows the details of one iteration of the repartitioner algorithm performed on a partition P_s. The algorithm detects the candidate vertices (Lines 4-8), selects the *top-k* candidates (Line 9), and moves them to their respective target partitions. Note that the migration in Line 11 is logical. After each phase of each iteration, the auxiliary data associated with each migrated vertex v is updated. This is because the neighbors of v may also be migrated, which would mean that the degree of v in each partition, i.e., auxiliary data associated with v, has changed. The algorithm continues moving vertices until there is no candidate vertex for migration, i.e., further movement of vertices does not improve edge-cut.

Example: To demonstrate the workings of the lightweight repartitioner, we show two iterations of the repartitioning algorithm on the graph of Figure 3 in which there are $\alpha = 3$ partitions and the average weight of partitions is 10/3. Assume the value of γ is $1.\bar{3}$. Hence, the aggregate weight of a partition needs to be in range $[2.\bar{2}, 4.\bar{4}]$; otherwise the partitioning is overloaded or underloaded. Figure 3a shows the

Algorithm 2 Lightweight Repartitioner

```
1: procedure REPARTITIONING_ITERATION(partition $P_s$)
2:     for stage $\in \{1, 2\}$ do
3:         $candidates \leftarrow \{\}$
4:         for Vertex $v \in$ VertexSet($P_s$) do
5:             $target(v) \leftarrow$ GET_TARGET_PART(v,stage)
6:                                 ▷ setting $target(v)$ and $gain(v)$
7:             if target(v) $\neq$ null then
8:                 candidates.add ($v$)
9:         $top\text{-}k \leftarrow k$ candidates with maximum gains
10:        for Vertex $v \in top\text{-}k$ do
11:            MIGRATE($v, P_S, target(v)$)
12:        $P_s$.update_auxiliary data
```

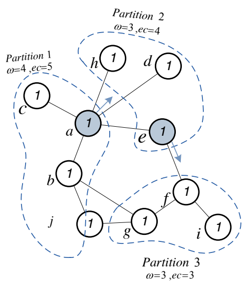
initial state of the graph. The partitions are sub-optimal as 6 of the 11 edges shown are edge-cuts. Consider the first stage of the first iteration of the lightweight repartitioner. Since the first stage restricts vertex migrations from lower ID partitions to higher ID only, vertices a and e are the migration candidates since they are the only ones that can improve edge-cut. Note that if the algorithm was performed in one stage, vertices h and d would be migrated to partition 1 causing the oscillating behavior discussed previously. At the end of the first stage of the first iteration, the state of the graph is as presented in Figure 3b. In the second stage, the algorithm migrates only vertex g. While vertex c could be migrated to improve edge-cut, the migration direction does not allow this (Figure 3c). In addition, such migration would cause partition 1 to be underloaded (its load will be 2 which is less than $2.\bar{2}$). In the second iteration, vertex c is migrated to partition 2. The result of the first stage of iteration 2 is presented in Figure 3d. At this point, the graph reaches an optimal grouping, thus the second stage of the second iteration will not perform any migrations. In fact further iterations would not migrate anything since the graph has an optimal partitioning.

3.2 Physical Data Migration

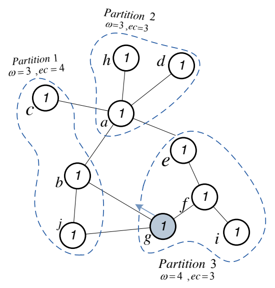
Physical data migration is the final step of the repartitioner. Vertices and relationships that were marked for migration by the repartitioner are moved to the target partitions using a two step process: (1) Copy marked vertices and relationships (copy step) (2) Remove marked vertices and relationships from the host partitions (remove step).

In the first step, a list of all vertices selected for migration to a partition are received by that partition, which will request these vertices and add them to its own local database. At the end of the first step, all moved vertices are replicated. Because of the insertion-only operations, the complexity of the operations is lower as all operations can be performed locally in each partition, meaning less network contention and locks held for shorter periods.

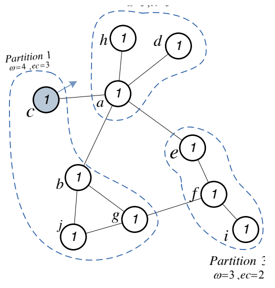
Between the two steps there is a synchronization process between all partitions to ensure that partitions have completed the copy process before removing marked vertices from their original partitions. The synchronization itself is not expensive as no locks or system resources are held, though partitions may need to wait until an occasional straggler finishes copying. In the remove step, all marked vertices will enter an unavailable state in which all queries referencing the vertex will be executed as if the vertex is not part



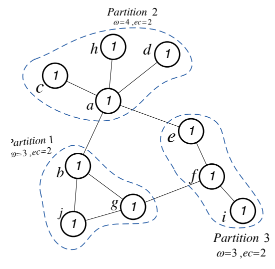
(a) Initial graph, before the first iteration



(b) After the first stage of the first iteration



(c) After the second stage of the first iteration



(d) After the first stage of the second iteration

Figure 3: Two iterations of the lightweight repartitioner. Two metrics are attached to every partition: ω representing the weight of the partition and ec representing the edge-cut.

of the local vertex set. This allows performing the transactional operations much faster as locks on unavailable vertices cannot be acquired by any standard queries.

3.3 Lightweight Repartitioner Analysis

3.3.1 Memory and Time Analysis

Recall that the main advantage of the lightweight repartitioner over multilevel algorithms is that it makes use of only auxiliary data to perform repartitioning. Auxiliary data has a small size compared to the size of the graph. This is formalized in the following two theorems, the proofs of which can be found in the extended version of the paper [25].

THEOREM 2. *The amortized size of auxiliary data stored on each partition to perform repartitioning is $n + \Theta(\alpha)$ on average. Here, n denotes the number of vertices in the input graph and α is the number of partitions.*

When compared to the multilevel algorithms, the memory requirement of the lightweight repartitioner is far less and can be easily maintained without hardly any impact on performance of the system. This is experimentally verified in Section 5.3.

THEOREM 3. *Each iteration of the repartitioning algorithm takes $O(\alpha n_s)$ time to complete. Here, α denotes the number of partitions and n_s is the number of vertices in the partition which runs the repartitioning algorithm.*

The above theorem implies that each iteration of the algorithm runs in linear time. Moreover, the algorithm converges to a stable partitioning after a small number of iterations relative to the number of vertices, e.g., in our experiments, it converges after less than 50 iterations, while there are millions of vertices in the graph data sets.

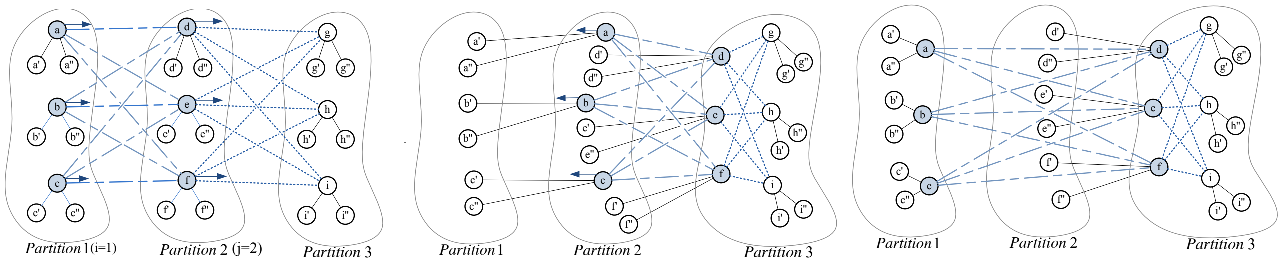
The lightweight repartitioner is designed for scalability and with little overhead to the database engine. The simplicity of the algorithm supports parallelization of operations and maximizes scalability. In the first phase, each iteration is performed in parallel on each server. The auxiliary data information is fully local to each server, thus lines 4 through 9 of Algorithm 2 are executed independently on each server. In the second phase of the repartitioning algorithm, physical data migration is performed. As mentioned in Section 2.2, this part has been decomposed into two steps for simplicity and performance. Because information is only copied in the first step (in which vertices are replicated), it allows for maximum parallelization with little need to synchronize between servers.

3.3.2 Algorithm Convergence

When the lightweight repartitioner triggers, the algorithm starts by migrating vertices from overloaded partitions. Note that no vertex is a candidate for migration to an overloaded partition. Hence, after a bounded number of iterations, the partitioning becomes valid in term of load balance. When there is no overloaded partition, the algorithm moves a vertex only if there is a positive gain in moving it from the source to the target partition. This is the main idea behind the following proof for the convergence of the algorithm.

THEOREM 4. *After a bounded number of iterations, the lightweight repartitioner algorithm converges to a stable partitioning in which further migration of vertices (as done by the algorithm) does not result in better partitionings.*

PROOF. We show that the algorithm constantly decreases the number of edge-cuts. For each vertex v, let $d_{ex}(v)$ denote the number of external neighbors of v, i.e., number of neighbors of v in partitions other than that of v. With this definition, the number of edge-cuts in a partition is $\chi/2$ where $\chi = \sum_{v=1}^{n} d_{ex}(v)$. Recall that the algorithm works in stages so that if in a stage migration of vertices is allowed from one partition to another, in the subsequent stage the migration is allowed in the opposite direction. We show that the value of χ decreases in every two subsequent stages; more precisely, we show that when a vertex v migrates in a stage t, the value of $d_{ex}(v)$ either decreases at the end of the stage t or at the end of the subsequent stage $t+1$ (compared to when v does not migrate). Let $d_k^t(v)$ denote the number of neighbors of vertex v in partition k before stage t. Assume that vertex v is migrated from partition i to partition j at stage t (see Figure 4). This implies that the number of neighbors of v in partition j is more than partition i. Hence, when v moves to partition j, the value of $d_{ex}(v)$ is expected to decrease. However, in a worst-case scenario, some neighbors of v in partition j also move to other partitions at the same stage (Figure 4b). Let $x(v)$ denote the number of neighbors of v in the target partition j which migrate at stage t; hence, at the end of the stage, the value of $d_{ex}(v)$ decreases by at least $d_j^t(v) - x(v)$ units. Moreover, $d_{ex}(v)$ is increased by at most $d_i^t(v)$; this is because the previous internal neighbors (those which remain at partition i) will become external after the migration of v. If $d_j^t(v) - x(v) > d_i^t(v)$, the value of $d_{ex}(v)$ decreases at the end of the stage and we are done. Otherwise, we say a *bad migration* occurred. In these cases,



(a) Original graph (b) After the first stage (c) After the second stage

Figure 4: The number of edge-cuts might increase in the first stage (in the worst case), but it decreases after the second stage. In this example, the number of edge-cuts is initially 18 (a); this increases to 21 after the first stage (b), and decreases to 15 at the end of the second stage (c).

assuming k is sufficiently large, in the subsequent stage $t+1$, v migrates back to partition i since there is a positive gain in such a migration (Figure 4c), and this results in a decrease of $d_i^{t+2}(v)$ and an increase of at most $d_j^t(v) - x(v)$ in $d_{ex}(v)$. Consequently, the net increase in d_{ex} after two stages is $(d_i^t(v) - (d_j^t(v) - x(v))) + (d_j^t(v) - x(v) - d_i^{t+2}(v)) = d_i^t(v) - d_i^{t+2}(v)$. Note that if v does not move at all, d_{ex} increases $d_i^t(v) - d_i^{t+2}(v)$ units after two stages. Hence, in the worst case, the net decrease in $d_{ex}(v)$ is at least 0 for all migrated vertices (compared to when they do not move). Indeed, we show that there are vertices for which the decrease in d_{ex} is strictly more than 0 after two consecutive stages. Assuming there are α partitions, these are the vertices which migrate to partition α [in stages where vertices move from lower ID to higher ID partitions] or partition 1 [in stages where vertices move from higher ID to lower ID partitions]. In these cases, no vertex can move from the target partition to another partition; so the actual decrease in $d_{ex}(v)$ is the same as the calculated gain when moving the vertex and is more than 0. To summarize, for all vertices, the value of $d_{ex}(v)$ does not increase after every two stages, and for some vertices, it decreases. For smaller values of k, after a bad migration, vertex v might not return from partition j to its initial partitioning i in the subsequent stage (since there might be more gain in moving other vertices); however, since there is a positive gain in moving v back to partition i, in subsequent stages, the algorithm moves v from partition j to another partition (i or another partition which results in more gain). The only exception is when many neighbors of v move to partition j so that there is no positive gain in moving v. In both cases, the value of $d_{ex}(v)$ decreases with the same argument as above. To conclude, as the algorithm runs, the accumulated values of $d_{ex}(v)$ (i.e., χ), and consequently the number of edge-cuts, constantly decrease. □

The graph structure in social networks does not evolve quickly and its evolution is towards community formation. Hence, as our experiments confirm, after a small number of iterations, the lightweight repartitioner converges to a stable partitioning. The speed of convergence depends on the value of k (the number of migrated vertices from a partition in each iteration). Larger values of k result in faster improvement on the number of edge-cuts and subsequently achieve partitioning with almost an optimal number of edge-cuts. However, as mentioned earlier, large values of k can degrade the balance factor of partitioning. Finding the right of value of k requires considering a few parameters which include the number of partitions, the structure of the graph (e.g., the average size of the clusters formed by vertices), and the nature of changing workload (whether the changes are mostly on the weight or on the degree of vertices). In practice, we observed that a sub-optimal value of k does not degrade convergence rate by more than a few iterations; consequently the algorithm does not require fine tuning for finding the best value of k. In our experiments, we set k as a small fraction of the number of vertices.

4. HERMES SYSTEM OVERVIEW

In this section, we provide an overview of Hermes, which we designed as an extension of Neo4j Version 1.7.3 to handle distribution of graph data and dynamic repartitioning. Neo4j is an open source centralized graph database system which provides a disk-based, transactional persistence engine (ACID compliant). The main querying interface to Neo4j is traversal based. Traversals use the graph structure and relationships between records to answer user queries.

To enable distribution, changes to several components of Neo4j were required as well as addition of new functionality. The modifications and extensions were done such that existing Neo4j features are preserved. Figure 5 shows the components of Hermes with the components of Neo4j that were modified to enable distribution in light blue shading while the components in dark blue shading are newly added. Detailed descriptions of the remaining changes are omitted as they pose technical challenges which were overcome using existing techniques. For example, as the centralized loop detection algorithm used by Neo4j for deadlock detection does not scale well, it was replaced using a timeout-based detection scheme as described in [10].

Internally, Neo4j stores information in three main stores: node store, relationship store and property store. Splitting data into three stores allows Neo4j to keep only basic information on nodes and relationships in the first two stores. Further, this allows Neo4j to have fixed size node and relationship records. Neo4j combines this feature with a monotonically increasing ID generator such that a) record offsets are computed in O(1) time using their ID and b) contiguous ID allocation allows records to be as tightly packed as possible. The property store allows for dynamic length records. To store the offsets, Neo4j uses a two layer architecture where a fixed size record store is used to store the offsets and

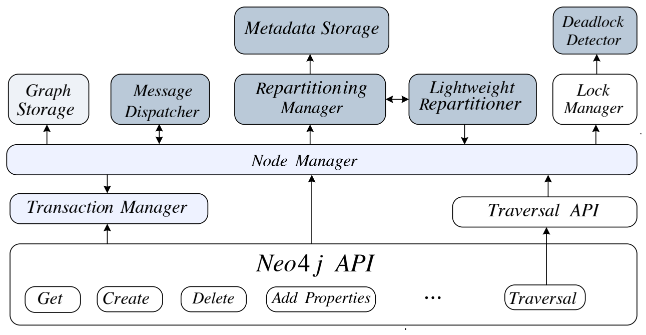


Figure 5: Hermes system layers together with modified and new components designed to make it run in a distributed environment.

a dynamic size record store is used to hold the properties. To shard data across multiple instances of Hermes, changes were made to allow local nodes and relationships to connect with remote ones. Hermes uses a doubly-linked list record model when keeping track of relationships. Such a node in the graph needs to know only the first relationship in the list since the rest can be retrieved by following the links from the first. Due to tight coupling between relationship records, referencing a remote node means that each partition would need to hold a copy of the relationship. Since replicating and maintaining all information related to a relationship would incur significant overhead, the relationship in one partition has a ghost flag attached to it to connect it with its remote counterpart. Relationships tagged by the ghost flag do not hold any information related to the properties of the relationship but are maintained to keep the graph structure valid. One advantage of this is the complete locality in finding the adjacency list of a graph node. This is important since traversal operations build on top of adjacency list.

The storage was also modified to use a tree-based indexing scheme (B+Tree) rather than an offset-based indexing scheme since record IDs can no longer be allocated in small increments. In addition, data migration would make offset based indexing impossible as records would need to be both compacted and still keep an offset based on their ID.

In Hermes, servers are connected in a peer-to-peer fashion similar to the one presented in Figure 6. A client can connect to any server and perform a query. Generally, user queries are in the form of a traversal. To submit a query the client would first lookup the vertex for the starting point of the query, then send the traversal query to the server hosting the initial vertex. The query is forwarded to the server containing the vertex such that data locality is maximized. On the server side, the traversal query will be processed by traversing the vertex's relationships. If the information is not local to the server, remote traversals are executed using the links between servers. When the traversal completes, the query results will be returned to the client.

5. PERFORMANCE EVALUATION

In this section, we present the evaluation of the lightweight repartitioner implemented into Hermes.

5.1 Experimental Setup

All experiments were executed on a cluster with 16 server

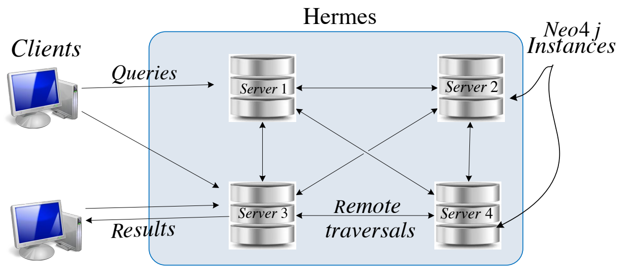


Figure 6: Overview of how Hermes servers interact with clients and with each other.

machines. Each server has the following hardware configuration: 2 AMD Opteron 252 (2 cores), 8 GB RAM and 160GB SATA HDD. The servers are connected using 1Gb ethernet. In each experiment, one Hermes instance runs on its own server.

The experiments are focused on typical social network traffic patterns, which based on previous work [8, 21], are 1-hop traversals and single record queries. We also consider 2-hop queries which are used for analytical queries such as ads and recommendations. Given the small diameters of social graphs (Table 1), queries with more than 2-hops are more typical of batch processing frameworks rather than social graphs where querying most or all of the graph data is required. The submission of traversal queries was described in Section 4.

5.2 Datasets

Three real-world datasets, namely Orkut, DBLP, and Twitter, are used to evaluate the performance of the lightweight repartitioner. We consider average path length, clustering coefficient, and power law coefficient of these graphs to characterize the datasets (Table 1). Average path length is the average length of the shortest path between all pairs of vertices. The clustering coefficient (a value between 0 and 1) measures how tightly clustered vertices are in the graph. A high coefficient means strong (or well connected) communities exist within the network. Finally, power law coefficient shows how the number of relationships increases as user popularity increases.

5.3 Experimental Results

The lightweight repartitioner is compared with two different partitioning algorithms. For an upper bound, we use a member of Metis family of repartitioners that is specifically designed for partitioning graphs whose degree distribution follows a power-law curve [6]. These graphs include social networks which are the focus of this paper.

Several previous partitioning approaches (e.g.[26, 28]) are

	Twitter	**Orkut**	**DBLP**
Number of nodes	11.3 million	3 million	317 thousand
Number of edges	85.3 million	223.5 million	1 million
Number of symmetric links	22.1%	100%	100%
Average path length	4.12	4.25	9.2
Clustering coefficient	unpublished	0.167	0.6324
Power law coefficient	2.276	1.18	3.64

Table 1: Summary description of datasets

compared against Metis as it is considered the "gold standard" for the quality of partitionings. It is also flexible enough to allow custom weights to be specified and used as a secondary goal for partitioning. We also compare the lightweight repartitioner against random hash-based partitioning, which is a de-facto standard in many data stores due to its decentralized nature and good load balance properties. Note that Metis is an offline, static partitioning algorithm that requires a very large amount of memory for execution. This means that either additional resources need to be allocated to partition and reload the graph every time the partitioner is executed, or the system has to be taken offline to load data on the servers. When the servers were taken offline, it took 2 hours to load each of the Orkut and Twitter graphs separately. This long period of time is unacceptable for production systems. Alternatively, if Hermes is augmented to run Metis on graphs, the resource overhead for running Metis would be much higher than the lightweight repartitioner. Metis' memory requirements scale with the number of relationships and coarsening stages, while the lightweight repartitioner scales with the number of vertices and partitions. Since the number of relationships dominates by orders of magnitude, Metis will require significantly more resources. For example, we found that Metis requires around 23GB and 17GB of memory to partition the Orkut and Twitter datasets, respectively; however, the lightweight repartitioner only requires 2GB and 3GB for these datasets. While Metis has been extended to support distributed computation (ParMetis [4]), the memory requirements for each server would still be higher than the lightweight repartitioner.

5.3.1 Lightweight Repartitioner Performance

Our experiments are derived from real world workloads [21, 8] and are similar to the ones in related papers [27, 24]. We first study 1-hop traversals on partitions with a randomly selected starting vertex. At the start of the experiments, the workload shifts such that the repartitioner is triggered, showing the performance impact of the repartitioner and the associated improvements. This shift in workload is caused by a skewed traffic trace where the users on one partition are randomly selected as starting points for traversals twice as many times as before, creating multiple hotspots on a partition. This workload skew is applied for the full duration of the experiments that follow.

Figure 7 presents the percentage of edge-cuts among all edges for both lightweight repartitioner and Metis on the skewed data. As the figure shows, the difference in edge-cut is too small (1% or less) to be significant, and we expect that this very small difference could shift in the other direction depending on factors such as query patterns and number of partitions. However, Figure 7 demonstrates that the lightweight repartitioner generates partitionings that are almost as good as those of Metis.

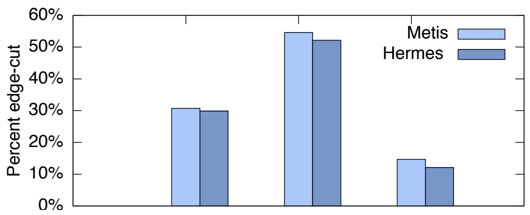
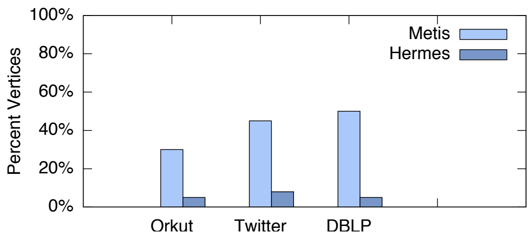
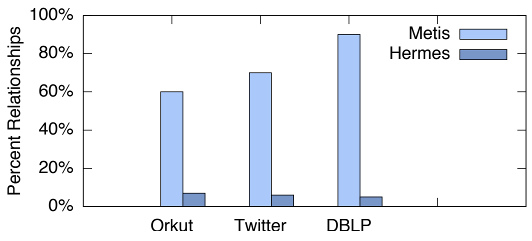


Figure 7: The number of edge-cuts in partitionings of the lightweight repartitioner (as a component of Hermes) versus Metis. Results are presented as a percentage of edge-cuts among the total number of edges.



(a) Migrated vertices.



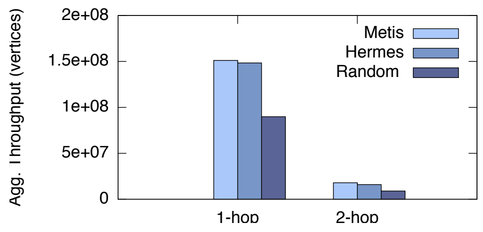
(b) Changed or migrated relationships.

Figure 8: The number of vertices (a) and relationships (b) changed or migrated as a result of the lightweight repartitioner (Hermes) versus running Metis.

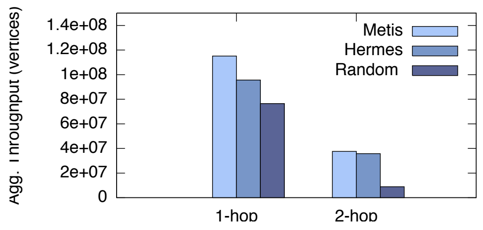
A repartitioner's performance is affected by the amount of data that it needs to migrate. To quantify the impact of migration on performance, the partitions resulting from the lightweight repartitioner and Metis are compared with the initial partitioning. Figure 8a shows the number of vertices migrated due to the skew based on the two partitioning algorithms. The results show a much lower count for the lightweight repartitioner. Figure 8b shows that the lightweight repartitioner requires, on average, significantly fewer changes to relationships compared to Metis. This difference is more extensive in the case of DBLP. The lightweight repartitioner is able to rebalance workload by moving 2% of the vertices and about 5% of the relationships, while Metis migrates an order of magnitude more data.

Overall, both the numbers of vertices and relationships migrated are important as they directly relate to the performance of the system. We note that, however, the relationship count has a higher impact on performance as this number will generally be much higher and relationship records are larger, and thus more expensive to migrate.

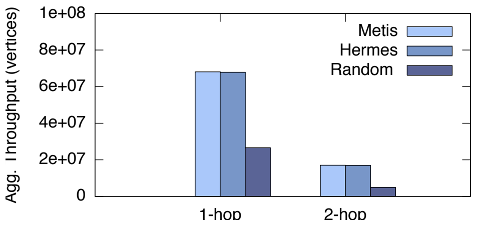
Figure 9 presents the aggregate throughput performance (i.e., the number of visited vertices) of 16 machines (partitions) using the three datasets. In these experiments, 32 clients concurrently submit 1-hop traversal requests using the previously described skew. Before the experiments start, Metis is applied to form an initial partitioning which has a trace with no skew so as to remove partitioning bias by starting out with a good partitioning. Once the experiment starts, the mentioned skew is applied; this skew triggers the repartitioning algorithm, whose performance is compared with running Metis *after* the skew. For Orkut, results show that by introducing the skew and triggering the lightweight repartitioner, a 1.7 times improvement in performance can



(a) Orkut



(b) Twitter



(c) DBLP

Figure 9: Aggregate throughput for the three datasets.

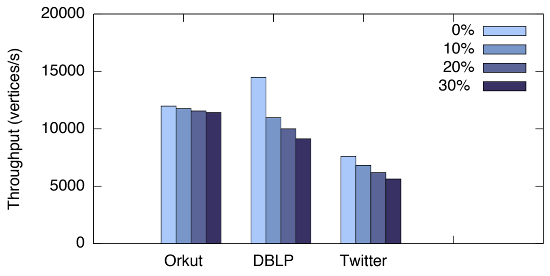


Figure 10: Throughput while varying the write rate.

be obtained over random partitioning while Metis shows a 6% improvement over the lightweight repartitioner. Figure 9b shows the aggregated throughput while running the Twitter dataset. The results show very similar performance for the lightweight repartitioner and Metis. Finally, Figure 9c shows the results related to the DBLP experiments that indicate there is no performance degradation due to the lightweight repartitioner, which benefits from the relatively small changes required by the algorithm. In fact, based on results from Figure 9c, the performance difference is not significant. Interestingly, the DBLP dataset is the only dataset for which the performance differences are not noticeable due to the highly clustered and well partitioned dataset. Given an edge-cut of 15%, the high query locality means that partition skews have little effect on performance as they do not shift workloads towards partition borders.

5.3.2 *2-hop Performance*

The previous section focused on 1-hop traversals . In this section, we conduct 2-hop experiments since they are representative operations used for recommendations, e.g., friend, events or ad recommendations in social networks. Figure 9 shows the aggregated performance of the system running with the three data-sets. The 2-hop experimental results are similar to 1-hop except for the decrease in performance. To analyze why this decrease occurs in the 2-hop case, we observe that the ratio between the number of vertices in the query response versus the number of vertices processed is 1 for both Metis and Random partitioners in case of 1-hop traversal queries, while these ratios degrade to 0.39 and 0.28, respectively, for 2-hop traversal queries. The reason 2-hop traversals return less vertices than what it processes is because some vertices are visited multiple times during a traversal while the query response contains only one copy of the queried vertices. Since social networks exhibit high clustering, a high fraction of processed vertices are accessed multiple times within the same traversal.

5.3.3 *Mixed Read/Write Experiments*

The following experiments test how the system handles mixed traffic workload and evolving social network graphs. The experiments insert data through random write traffic. The lightweight repartitioner in Hermes is then run to improve the quality of partitioning after records are inserted. Results of these experiments are shown in Figure 10 and indicate little performance degradation with increasing write traffic. A 10% write mix (with 90% reads) show a 3% decrease in performance, while 20% writes (80% reads) and 30% writes (70% reads) show 5% and 7% decreases in throughput (vertices per second) performance. The small performance impact of writes is attributed to how B+Trees store information and the monotonically increasing ID generator in Hermes. Since each new record will get the next, highest ID, insertions in the B+Tree always happen in the last page in a sequential manner. This translates to sequential writes to disk and the B+Tree requires caching only the last page to perform insertions. To verify that the quality of the graph is high after the insertions finish, we ran 100% read traffic and compared the throughput with the results for Metis. Results showed that Hermes was able to keep partition quality and system performance within 2% of Metis, which demonstrates the effectiveness and efficiency of Hermes's lightweight repartitioner.

5.3.4 *Sensitivity of Repartitioner Parameters*

Recall from Section 3 that in each iteration of the algorithm, the lightweight repartitioner moves at most k vertices from each partition. Here, we examine how the value of k affects the outcome of the algorithm. We run the lightweight repartitioner with three different values of k (500, 1000, and 2000). The first observation is that the load-balance factor slightly degrades from 1.05 for $k = 500$ to 1.16 for $k = 2000$. This is because, as mentioned earlier, larger values of k result in simultaneous migration of many vertices to partitions which have recently become popular (due to hosting popular vertices). Consequently, we excluded values of k large than 2000 from the experiment as they result in imbalance factor more than the maximum allowed value of $\gamma = 1.1$ (the default value of γ in the system). Next, we verified how

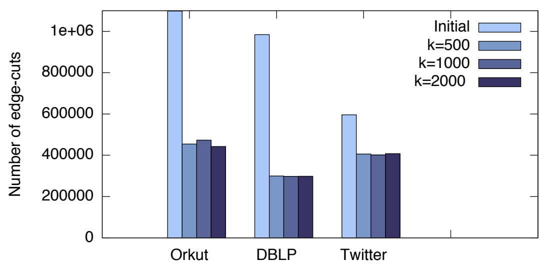


Figure 11: The number of edge-cuts for different values of k. The numbers are scaled by 10^{-2} for Orkut dataset.

many iterations are required for the algorithms to converge. Table 2 shows the number of required iterations for different values of k. As expected, larger values of k result in slightly faster convergence since they move more vertices per iteration. Finally, we considered the quality of partitioning when different values of k are used. As Figure 11 shows, the number of edge-cuts in the final partitioning is almost the same for different values of k, indicating that the quality of partitioning does not depend on this value. To summarize, larger values of k result in faster convergence while increasing the load imbalance; at the same time, the value of k does not have a significant effect on the number of edge-cuts.

6. RELATED WORK

Previous work on graph databases focused on a centralized approach [17, 23, 1]. Some of these systems, e.g., Neo4j, have a high availability mode but this provides limited scalability [2]. HypergraphDB [17] provides a message passing API between server instances, however there is no partition management system and no support for distribution-aware querying. In addition, the focus of each of these systems is different. For example HyperGraphDB focuses more on the flexibility of its storage system, allowing users to store different types of objects. None of these graph databases support data partitioning or distributed graph querying.

SPAR [27] and Titan [5] are middleware that run on top of key-value stores or relational databases to provide on-the-fly partitioning and replication of data. However, SPAR is restricted to keeping only one-hop neighbours local while Hermes can support general remote traversals. Titan uses only static hash-based, random partitioning scheme supported by the underlying key-value store. This is in contrast to the dynamic repartitioning that the lightweight repartitioner in Hermes uses.

Unlike our system, SEDGE [36] focuses on partition replication in which a coarsening stage aggregates nodes which are then matched with nodes in another partition. SEDGE is not designed to handle dynamic workload changes that Hermes is designed for. An approach that performs dynamic replication is described in [24] but it does not involve a system that does on-the-fly partitioning. Horton [29] is a query execution engine built for distributed in-memory graphs. However it only provides a user abstraction, leaving partitioning to existing offline algorithms.

A streaming algorithm using simple heuristics has been proposed in [32] but focuses on improving initial data placement unlike the dynamic repartitioning in Hermes. In [33] an improved heuristic for better partitioning quality is proposed; however, the partition imbalance in the resulting solutions can be significantly impacted. While this approach extends the concept by saving state and allowing future data loads to reuse state from previous runs, the algorithm needs to parse the full dataset again, which can lead to expensive operations and large migrations.

Several Pregel-like [22] systems, e.g., [11, 16], have been proposed. These systems are quite different from Hermes in that they address only in-memory batch processing of graph analytics queries rather than the persistent management of graph data that Hermes is designed to support. In [35], a weighted multi-level partitioning algorithm is proposed which is based on label propagation (community detection technique). The edge-cut decrease in this approach can be small while communities might be detected improperly because of the weighted approach. Their multi-level algorithm does not guarantee communities are preserved over multiple calls of the algorithm and can lead to large migrations similar to Metis.

Some graph partitioning algorithms are experimentally compared in [9]. Among these, DiDiC [14] is the only distributed algorithm that tends to minimize the number of edge-cuts, but the resulting partitions of this approach may not be well-balanced [28, 9].

Ja-Be-Ja [28] embeds a distributed algorithm for balanced partitioning without global knowledge. In this algorithm, the initial partition of each node is selected uniformly at random; this ensures a balanced partitioning. In order to decrease the number of edge-cuts, vertices are swapped between partitions. This will ensure maintaining a balanced partitioning if vertices have fixed, uniform weights; however, this is usually not the case for social networks.

An adaptive algorithm for repartitioning large-scale graphs has the objective of minimizing the number of edge-cuts with respect to certain capacities for partitions [34]. The resulting partitionings might not be balanced if the capacity constraints are maintained. Moreover, it is assumed that vertices have fixed and uniform weights, which is usually not the case for social networks. Additionally, their work is targeted for graph analytics rather than the persistent management of graph data that Hermes is designed to support.

	Twitter	**Orkut**	**DBLP**
k = **500**	30	30	40
k = **1000**	17	17	13
k = **2000**	10	10	11

Table 2: The number of iterations after which the lightweight repartitioner converges.

7. CONCLUSION

We presented an online, iterative, lightweight repartitioner designed to increase query locality, thereby decreasing network load and maintaining load balance across partitions. The lightweight repartitioner effectively adapts a partitioning to varying query workloads and the continuous evolution of the graph structure using only a small amount of auxiliary data. We implemented our lightweight repartitioner into Hermes, which we built to extend the open source Neo4j database system to support the partitioning of social network data across multiple database servers. The experimental evaluation of the algorithm on real-world datasets shows that the repartitioner is able to handle changes in query workloads while maintaining good performance. The overhead of the repartitioner is minimal, producing sustained performance comparable to that of static, offline, partitioning using Metis. Our evaluation shows sizable performance gains over random, hash-based partitioning, which is widely used for database partitioning.

8. REFERENCES

[1] Neo4j. http://www.neo4j.org/.

[2] Neo4j - chapter 26. high availability. http://docs.neo4j.org/chunked/stable/ha.html.

[3] Neotechnology. http://www.neotechnology.com/customers/.

[4] Parmetis. http://glaros.dtc.umn.edu/gkhome/metis/parmetis /overview.

[5] Titan. http://thinkaurelius.github.com/titan/.

[6] Abou-Rjeili, A., and Karypis, G. Multilevel algorithms for partitioning power-law graphs. In *proc. IPDPS* (2006), pp. 124–124.

[7] Andreev, K., and Räcke, H. Balanced graph partitioning. *Theo. Comput. Syst. 39*, 6 (2006), 929–939.

[8] Armstrong, T. G., Ponnekanti, V., Borthakur, D., and Callaghan, M. Linkbench: a database benchmark based on the facebook social graph. In *proc. SIGMOD* (2013), pp. 1185–1196.

[9] Averbuch, A., and Neumann, M. Partitioning graph databases. Master's thesis, KTH School of Computer Science and Communication, 2013.

[10] Bernstein, P. A., and Newcomer, E. *Principles of Transaction Processing*, 2nd ed. 2009.

[11] Chen, R., Yang, M., Weng, X., Choi, B., He, B., and Li, X. Improving large graph processing on partitioned graphs in the cloud. In *proc. SoCC* (2012), pp. 1–13.

[12] Fiduccia, C. M., and Mattheyses, R. M. A linear-time heuristic for improving network partitions. In *proc. DAC* (1982), pp. 175–181.

[13] Garey, M. R., and Johnson, D. S. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* 1990.

[14] Gehweiler, J., and Meyerhenke, H. A distributed diffusive heuristic for clustering a virtual P2P supercomputer. In *proc. IPDPS* (2010), pp. 1–8.

[15] Golbeck, J. *Analyzing the Social Web.* Elsevier Inc., 2013.

[16] Han, M., Daudjee, K., Ammar, K., Özsu, M. T., Wang, X., and Jin, T. An experimental comparison of pregel-like graph processing systems. *PVLDB 7*, 12 (2014), 1047–1058.

[17] Iordanov, B. Hypergraphdb: A generalized graph database. In *proc. WAIM Workshops* (2010), pp. 25–36.

[18] Karypis, G., and Kumar, V. Metis - unstructured graph partitioning and sparse matrix ordering system (vers. 2). Tech. rep., Univ. of Minnesota, 1995.

[19] Karypis, G., and Kumar, V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput. 20*, 1 (1998), 359–392.

[20] Kernighan, B. W., and Lin, S. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell system technical journal 49*, 1 (1970), 291–307.

[21] Kwak, H., Lee, C., Park, H., and Moon, S. What is Twitter, a social network or a news media? In *Proc. WWW* (2010), pp. 591–600.

[22] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. Pregel: a system for large-scale graph processing. In *proc. SIGMOD* (2010).

[23] Martínez-Bazan, N., Muntés-Mulero, V., S. Gómez-Villamor, S., Nin, J., Sánchez-Martínez, M., and Larriba-Pey, J. Dex: high-performance exploration on large graphs for information retrieval. In *proc. CIKM* (2007).

[24] Mondal, J., and Deshpande, A. Managing large dynamic graphs efficiently. In *proc. SIGMOD* (2012), pp. 145–156.

[25] Nicoara, D., Kamali, S., Daudjee, K., and Chen, L. Hermes: Dynamic partitioning for distributed social network graph databases. Tech. Rep. CS-2015-01, School of Computer Science, University of Waterloo, 2015.

[26] Nishimura, J., and Ugander, J. Restreaming graph partitioning: simple versatile algorithms for advanced balancing. In *proc. KDD* (2013), pp. 1106–1114.

[27] Pujol, J., Erramilli, V., Siganos, G., Yang, X., Laoutaris, N., Chhabra, P., and Rodriguez, P. The little engine(s) that could: scaling online social networks. *SIGCOMM CCR 40*, 4 (2010), 375–386.

[28] Rahimian, F., Payberah, A. H., Girdzijauskas, S., Jelasity, M., and Haridi, S. JA-BE-JA: A distributed algorithm for balanced graph partitioning. In *proc. SASO* (2013), pp. 51–60.

[29] Sarwat, M., Elnikety, S., He, Y., and Kliot, G. Horton: Online query execution engine for large distributed graphs. In *proc. ICDE* (2012).

[30] Schloegel, K., Karypis, G., and Kumar, V. Multilevel diffusion schemes for repartitioning of adaptive meshed. *TR 97-013, U. Minnesota* (1997).

[31] Schloegel, K., Karypis, G., and Kumar, V. Multilevel diffusion schemes for repartitioning of adaptive meshes. *J. Parallel Distrib. Comput. 47*, 2 (1997), 109–124.

[32] Stanton, I., and Kliot, G. Streaming graph partitioning for large distributed graphs. In *proc. KDD* (2012), pp. 1222–1230.

[33] Tsourakakis, C. E., Gkantsidis, C., Radunovic, B., and Vojnovic, M. Fennel: Streaming graph partitioning for massive scale graphs. In *proc. WSDM* (2014).

[34] Vaquero, L. M., Cuadrado, F., Logothetis, D., and Martella, C. Adaptive partitioning for large-scale dynamic graphs. In *proc. ICDCS* (2014), pp. 144–153.

[35] Wang, L., Xiao, Y., Shao, B., and Wang, H. How to partition a billion-node graph. In *proc. ICDE* (2014).

[36] Yang, S., Yan, X., Zong, B., and Khan, A. Towards effective partition management for large graphs. In *proc. SIGMOD* (2012), pp. 517–528.

On Debugging Non-Answers in Keyword Search Systems

Akanksha Baid Wentao Wu Chong Sun AnHai Doan Jeffrey F. Naughton
Department of Computer Sciences, University of Wisconsin-Madison
1210 W. Dayton St., Madison, WI, USA
{baid, wentaowu, sunchong, anhai, naughton}@cs.wisc.edu

ABSTRACT

Handling non-answers is desirable in information retrieval systems. Current e-commerce websites usually try to suppress the somewhat dreaded message that no results have been found. Possible solutions include, for example, augmenting the data with synonyms and common misspellings based on query logs. Nonetheless, this is only achievable if we can know the cause of the non-answers. Under the hood, most e-commerce data sits in some *structured* format. Debugging non-answers in the underlying KWS-S systems is therefore not trivial — non-answers in a KWS-S system could be a problem of the data (e.g., absence of some keywords), the schema (e.g., missing key-foreign-key joins), or due to empty join results from one of possibly several joins in the generated SQL queries. So far, we are unaware of any previous work that explores how to enable developers to debug non-answers in a KWS-S system. In this paper, we take a first step towards this direction by proposing a KWS-S system that can expose non-answers to the developers. Our system presents the developers with the maximal nonempty sub-queries that represent the frontier cause of the non-answers. We outline the challenges in building such a system and propose a lattice structure for efficient exploration of the non-answer query space. We also evaluate our proposed mechanisms over a real world dataset to demonstrate their feasibility.

1. INTRODUCTION

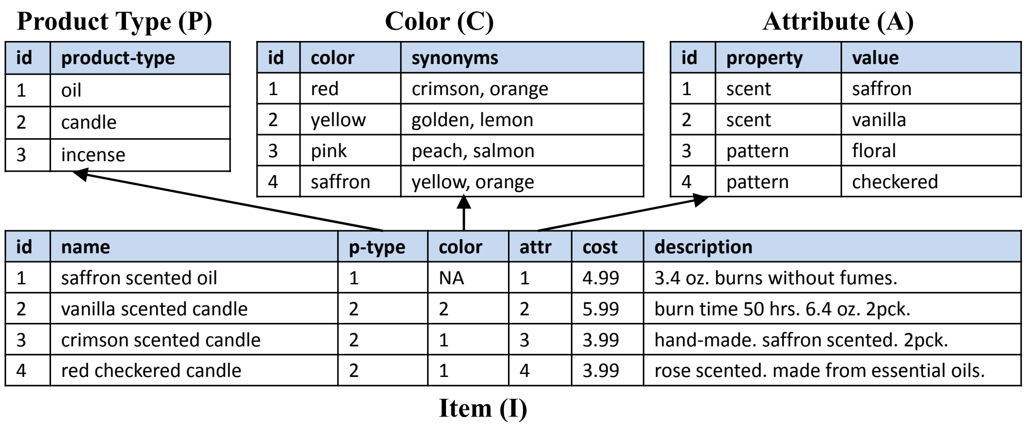
Handling non-answers (i.e., queries that return no results) is now a common practice in information retrieval systems. Current SEO companies and e-commerce websites like Orcale Endeca [21], HP Autonomy [9], and IBM Coremetrics [13] often try to avoid showing the somewhat dreaded "*No results found*!" message when they fail to return any results that can match user's keyword queries. Possible strategies include, for instance, substituting user's original keywords with different keywords from a controlled vocabulary (e.g., synonyms, hyponyms, and hypernyms), or displaying a "*Did you mean*?" style response with spelling corrections. Doing so is critical to helping customers find what they are looking for and improving user experience and ultimately retention.

Nonetheless, implementation of such seemingly simple strategies is not trivial. While users interact with a search box, under the hood most e-commerce data sits in some *structured* format, largely due to maintainablity reasons. To employ strategies such as augmenting the data with synonyms and common misspellings based on query logs, we first need to understand the cause of the non-answers. For example, we need to convince ourselves that a non-answer query is really caused by missing keywords in the data before we decide to add the missing words into the vocabulary; otherwise this action is not helpful. Unfortunately, debugging non-answers in the underlying KWS-S (acronym for KeyWord Search over Structured data) systems is challenging — non-answers in a KWS-S system could be a problem of the data (e.g., absence of some keywords), the schema (e.g., missing key-foreign-key joins), or due to empty join results from one of possibly several joins in the generated SQL queries. So far, we are unaware of any previous work that explores how to enable developers to debug non-answers in a KWS-S system.

In this paper, we take a first step towards systematically exploring non-answers in KWS-S systems, and we focus on seeking the *maximal* partial matches or sub-queries of the non-answers. Similar ideas have been explored in the *unstructured* world. For example, Figure 1 presents the screenshot from buy.com in response to the keyword query "`saffron scented candle`". Although no saffron-scented candles are found, rather than displaying a blank page that shows no results, other saffron-scented products and other scented candles are presented to the user, corresponding to the three sub-queries "`saffron scented`", "`saffron candle`", and "`scented candle`". We believe that this kind of information could also be very helpful for debugging non-answers in KWS-S systems, and our primary goal in this paper, akin to what has been done over unstructured data, is to find results from sub-queries of non-answers, but over *structured* data.

Moving from unstructured to structured data, however, is more complicated than we might have thought. In typical KWS-S systems such as Banks [1, 14], DBXplorer [2], and DISCOVER [11], users enter a set of keywords and the system responds with a multitude of relationships connecting those keywords. In [2, 11, 17, 19] and many other KWS-S systems, this is done by mapping the keyword query to several structured queries (i.e., SQL queries). All of these structured queries are then evaluated, and the tuples corresponding to the queries that produce answers are returned to the user. Sub-queries of a non-answer query in a KWS-S system thus refer to sub-queries of a structured SQL query rather than the original keyword query. Since a KWS-S system can usually generate many SQL queries in response to a single keyword query, naively evaluating all possible sub-queries at runtime could be quite expensive. To efficiently explore the space of sub-queries, our basic idea is to exploit the common sub-queries that are shared by

10.5441/002/edbt.2015.05



Product Type (P)

id	product-type
1	oil
2	candle
3	incense

Color (C)

id	color	synonyms
1	red	crimson, orange
2	yellow	golden, lemon
3	pink	peach, salmon
4	saffron	yellow, orange

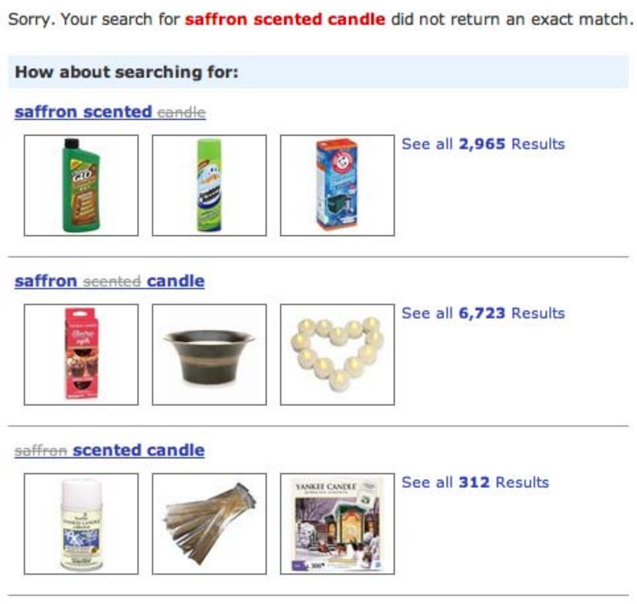
Attribute (A)

id	property	value
1	scent	saffron
2	scent	vanilla
3	pattern	floral
4	pattern	checkered

id	name	p-type	color	attr	cost	description
1	saffron scented oil	1	NA	1	4.99	3.4 oz. burns without fumes.
2	vanilla scented candle	2	2	2	5.99	burn time 50 hrs. 6.4 oz. 2pck.
3	crimson scented candle	2	1	3	3.99	hand-made. saffron scented. 2pck.
4	red checkered candle	2	1	4	3.99	rose scented. made from essential oils.

Item (I)

Figure 2: Product database containing an `Items` Table (I), `Product Type` table (P), `Color` table (C) and `Attributes` table (A).



Sorry. Your search for **saffron scented candle** did not return an exact match.

How about searching for:

saffron scented ~~candle~~

See all **2,965** Results

saffron ~~scented~~ candle

See all **6,723** Results

~~saffron~~ scented candle

See all **312** Results

Figure 1: Screenshot from buy.com where sub-queries and their results are suggested to the user when "saffron scented candles" returns zero products.

multiple structured SQL queries. To put things in context, let us consider the following example:

EXAMPLE 1 (NON-ANSWERS IN KWS-S). *Figure 2 shows a toy database that will be used throughout this paper. It contains an* `Items` *table I, a* `Product Type` *table P, a* `Colors` *table C, and an* `Attributes` *table A. The arrows here present the key-foreign-key associations between the tables.*

Consider the keyword query "`saffron scented candle`*". The KWS-S system maps it to two structured SQL queries (R^k here means the keyword k is mapped to the table R):*

(q_1) $P^{candle} \bowtie I^{scented} \bowtie C^{saffron}$, which tries to "find scented candles whose <u>*color*</u> *is saffron."*

(q_2) $P^{candle} \bowtie I^{scented} \bowtie A^{saffron}$, which tries to "find scented candles whose <u>*scent*</u> *is saffron."*

Both q_1 and q_2 return no result tuples with the given database, namely, they are non-answers. In the case of q_1, while every keyword does occur in the database, the join of all the involved tables produces no results. Exposing this information and q_1 can allow the developer or SEO person to add `saffron` *as a synonym of* `yellow`*, thus returning several relevant results to the user. In existing KWS-S systems q_1 would never be exposed.*

As for q_2, its sub-queries, which are $P^{candle} \bowtie I^{scented}$ and $I^{scented} \bowtie A^{saffron}$, do return answers, even though q_2 does not. More specifically, while the merchant does not carry any saffron-scented candles, it does carry scented candles and saffron-scented products that are not candles. Knowing this information may not help return answers to the original query, like in the q_1 case. However, it could be useful for merchandizing purposes. Additionally (like in Figure 1), the partial queries serve as a good alternative to returning nothing.[1]

Inspired by Chapman and Jagadish [5], to explain causes of the non-answers, we report their *maximal* sub-queries that return at least one tuple. To illustrate, in Example 1, our system will display $P^{candle} \bowtie I^{scented}$ and $C^{saffron}$ for q_1, and $P^{candle} \bowtie I^{scented}$ and $I^{scented} \bowtie A^{saffron}$ for q_2. Intuitively, these sub-queries sit on the *boundary* of answers/non-answers and provide the developer with information about the *frontier* causes of the non-answers. Similar notions have been proposed in previous work as solutions to "why not" style questions. For instance, in [5] the authors proposed using *frontier picky manipulations* which are the highest operators in a query tree or the latest manipulations in a workflow that rule out data items interested by the user from the results.

To find the maximal nonempty sub-queries for the non-answers, a naive strategy could be to enumerate all sub-queries and evaluate them (i.e., run the SQL query over the database) to check if they are empty. This is clearly inefficient. Our key observation here is that sub-queries of the non-answers overlap. For example, the q_1 and q_2 in Example 1 share the common join query $P^{candle} \bowtie I^{scented}$. In our experiments, we found that this overlap is significant on real data. Motivated by this observation, we propose a lattice structure that represents all the structured queries that a KWS-S system explores (details in Section 2). This structure is constructed offline and is used to capture the overlap between the sub-queries of each query (Section 2.2). Additionally, it also lends itself to systematic exploration of the sub-queries of non-answer queries. Note that, once we know the status of a query (i.e., if it is empty),

[1]The situation here is a bit more symmetric than that described in this example: given another instance of the tables, it might be q_2 that can be fixed via a synonym, whereas q_1 might be the query where non-answers are explained via maximal sub-queries.

this information can be utilized to determine the status of other queries based on the hierarchical relationships between queries presented in the lattice. For instance, in Example 1, we do not need to run the two SQL queries corresponding to P^{candle} and $I^{scented}$ once we know $P^{candle} \bowtie I^{scented}$ is nonempty — they must both be nonempty as well. This raises the interesting question of in which order we should visit the nodes in the lattice with the purpose of minimizing the number of SQL queries that need to be executed, which is the key to runtime system performance. We studied both *top-down* and *bottom-up* strategies to traverse the lattice structure (Section 2.5), and found that their performance depends on the distribution of the non-answer sub-queries within the hierarchy. Specifically, top-down/bottom-up strategies are more efficient when the maximal nonempty sub-queries are at higher/lower levels of the lattice. With this in mind, we further propose a greedy algorithm based on a scoring function that measures the potential reduction in the search space from examining a certain sub-query (Section 2.5.3). Our experimental results show that, while top-down and bottom-up strategies suffer from certain distributions of the non-answers, the greedy algorithm can perform relatively well in all the cases we tested.

While our proposed framework can efficiently find all the maximal non-empty sub-queries for non-answers, in our experiments we observed that the number of sub-queries is sometimes large. This is actually an inherent problem of KWS-S systems. Existing KWS-S systems usually use ranking functions to present users with only the most relevant results. For instance, Hristidis et al. [10] studied the problem of efficiently presenting the end users with a list of top-k matches. However, such strategies cannot work for the goal of debugging non-answers in KWS-S systems. This is simply because of the nature of debugging, which needs to find the cause of the non-answers no matter how trivial the cause might be. It is akin to debugging a normal computer program, where *all* possible bugs should be reported. Of course, there is a number of possible solutions to alleviate the problem of overwhelming number of sub-queries. For example, one option could be to allow the developer to define various filters or a priority hierarchy on the returned sub-queries. We do not try to explore all of these possible postprocessing techniques in this paper, most of which are application-specific and therefore may not have a uniformly optimal solution. Rather, we focus on an essential foundational task that must be solved before any higher-level postprocessing can be performed: the task of efficiently finding non-answers in response to a keyword query over structured data. It is our hope that our solution for this task provides a building block that can be used in conjunction with future research to build more customized systems.

In the rest of the paper, we start by introducing the system architecture of the proposed system and detailing its components in Section 2. We then evaluate our proposed approaches in Section 3. We discuss related work in Section 4 and conclude in Section 5.

2. EXPLORING NON-ANSWERS

In this section we describe our proposed solution for efficiently determining and explaining non-answer queries. Figure 3 presents the proposed system in its entirety. **Phase 0** is performed offline. In this phase, based on the schema graph of the underlying database, we generate a lattice in which each node is labeled with an uninstantiated SQL query. This structure is designed to exploit the overlap between the queries that are explored by our system. Following this, in **Phase 1** user's keyword query is accepted and used to prune the lattice generated in **Phase 0**. At the end of **Phase 1** each node in the pruned lattice is labeled with an instantiated SQL query with respect to the keyword query. In **Phase 2** we prune the lattice even further by retaining only those nodes that contain answer queries or non-answer queries, with respect to the current keyword query, and their respective descendants. Finally, we traverse the pruned lattice to determine and explain non-answer queries in **Phase 3**. Based on the information obtained, the user can subsequently choose to modify the keyword query as needed. We start by providing a formal problem definition describing the input and output of the proposed system.

2.1 Problem Definition

The input to our system is the keyword query submitted by the user. We convert the keyword query to join networks of tuple sets and candidate networks (see DISCOVER [11] for definitions).

The output of our system contains three parts: (i) answer queries, i.e., candidate networks that return at least one tuple; (ii) non-answer queries, i.e., candidate networks that return no tuples; (iii) additionally, for non-answer queries, we return the *maximal* sub-networks (i.e., subgraphs) that return at least one tuple. This is analogous to the queries in Figure 1, and is meant to provide some insight into the reasons behind the non-answer queries.

More formally, let J be a join network of tuple sets (JNTS) of a keyword query K. Let $q(J)$ be the SQL query corresponding to J and $R(J)$ be the result set of tuples obtained by executing $q(J)$.

Let $\mathcal{C}(K)$ be the set of candidate networks (CNs) generated for K. For each $C \in \mathcal{C}(K)$, we say that C is an *answer query* of K if $R(C) \neq \emptyset$. Otherwise C is a *non-answer query*. We denote $\mathcal{A}(K)$ and $\mathcal{N}(K)$ as the sets of answer and non-answer queries of K.

For each $C \in \mathcal{N}(K)$, let $\mathcal{S}(C)$ be the set of JNTSs that are sub-networks of C. A $J \in \mathcal{S}(C)$ is said to be *maximal* if: (i) $R(J) \neq \emptyset$ and (ii) there is no $J' \in \mathcal{S}(C)$ s.t. J is a sub-network of J' and $R(J') \neq \emptyset$. We use $\mathcal{M}(C)$ to denote the *maximal* JNTSs in $\mathcal{S}(C)$ and $\mathcal{M}(K)$ to denote the set of maximal JNTSs for all the non-answer queries, i.e., $\mathcal{M}(K) = \bigcup_{C \in \mathcal{N}(K)} \mathcal{M}(C)$.

With the above definitions and notation, we can now formally define the input and output of our system as follows:

- **Input**: An unstructured keyword query K.
- **Output**: $\mathcal{O}(K) = \mathcal{A}(K) \bigcup \mathcal{N}(K) \bigcup \mathcal{M}(K)$.

2.2 Offline Lattice Generation (Phase 0)

Phase 0 of the system is performed offline. In this phase we generate a lattice-structure that serves as the starting point for every keyword query. The goal of this structure is to capture all the queries that a KWS-S system explores (i.e., join-queries that contain no projections). Each node in the lattice is labeled with the SQL query corresponding to the node. The base level nodes of the lattice contain the simplest queries — single table queries, one for each table. The next level is generated by joining in tables to each single-table query (avoiding cross products and using the joins that are implicit in the schema graph), and so forth.

Our goal is to cover all queries with up to m joins. Since each relation can appear many times in a single query, we maintain copies $R_1 \ldots R_{m+1}$ of each relation R. In this way we know we can generate all possible m-join queries (including the extreme case where a m-join query contains $m+1$ instances of the same relation). In addition to this we also maintain a copy R_0 of every relation R in the database (explained in the next section).

EXAMPLE 2 (LATTICE). *Consider a database with only two relations $R(a, b)$ and $S(c, d)$. Assume that $m = 1$, namely, we allow only one key-foreign-key join $R.b \bowtie S.c$. As a result, the lattice contains two (i.e., $m + 1 = 1 + 1 = 2$) copies for each relation (except for the special R_0 and S_0).*

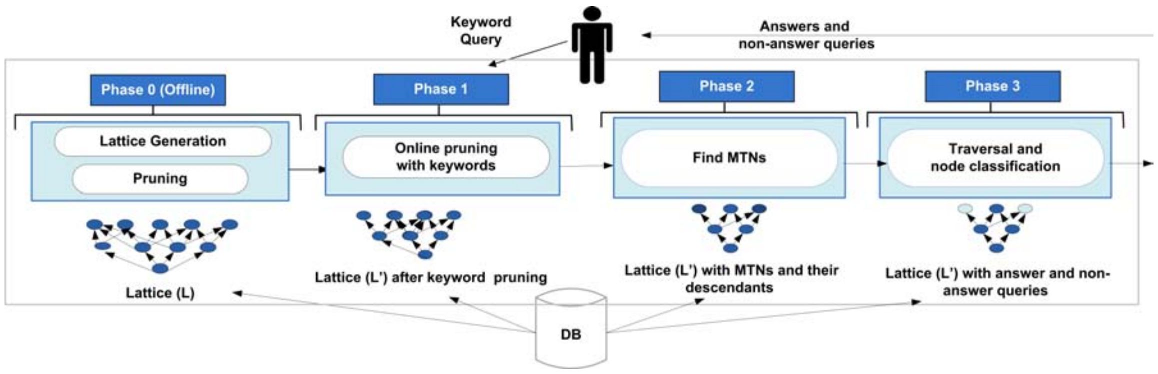

Keyword Query
Answers and non-answer queries
Phase 0 (Offline)
Phase 1
Phase 2
Phase 3
Lattice Generation
Pruning
Online pruning with keywords
Find MTNs
Traversal and node classification
Lattice (L)
Lattice (L') after keyword pruning
Lattice (L') with MTNs and their descendants
Lattice (L') with answer and non-answer queries
DB

Figure 3: System Architecture for the proposed system. (MTN means Minimal-Total Node: see Section 2.4.)

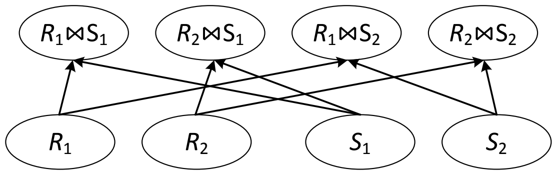


Figure 4: Example lattice with two relations $R(a,b)$ and $S(c,d)$, and a schema graph containing only one key-foreign-key join $R.b \bowtie S.c$.

Let "k_1 k_2" be the user's keyword query. As shown in Figure 4, the generated lattice has two levels. Additionally, each node in the lattice is bound to a SQL query template. For instance, the node $R_1 \bowtie S_2$ corresponds to the template

```
SELECT * FROM R1, S2 WHERE R1.b = S2.c
AND R1.a LIKE '%k1%' AND S2.d LIKE '%k2%'.
```

Note that, the copies here are just conceptual symbols rather than physical replicas. The purpose of introducing the copies is to maintain a 1-1 mapping between lattice nodes and SQL query templates, which reduces the run-time query processing overhead. If, on the other hand, no additional copies were maintained, then the lattice in Figure 4 can be reduced to containing only three nodes R, S, and $R \bowtie S$. While this could reduce the storage overhead, each node in the lattice would correspond to multiple SQL query templates, and the parent-child relationships between the nodes would need to be reconstructed at run-time. This would adversely impact the time required to process keyword queries.

Furthermore, if a node N in the lattice is a descendant of a node N', then the query in N is a sub-query of the query in N'. The lattice structure hence organizes the queries and sub-queries that in a hierarchical fashion. As we will see, this structure has three primary advantages — (i) it allows reuse of evaluated queries; (ii) sometimes, it allows us to infer the outcome of a SQL query without executing it; and (iii) its hierarchical structure allows us to systematically explore sub-queries, which can be used to better understand non-answer queries. Also, since this structure is computed offline, it bypasses the costly candidate network generation phase, which is a part of traditional KWS-S systems.

While offline processing allows us to generate all the combinations of join-queries without taking a performance hit at run-time, this process leaves us with many duplicates. The duplicates are due to the fact that a node in the lattice can be obtained by different extensions of its children. For example, in Figure 4, the node $R_1 \bowtie S_2$ can be obtained by either extending the node R_1 (joining it with S_2) or extending the node S_2 (joining it with R_1). We therefore need to eliminate as many duplicates as possible offline, to avoid expensive graph isomorphism tests at run-time. Eliminating duplicate nodes also helps with reuse. We talk more about reuse in Section 2.5.2.

Algorithm 1: Lattice Generation

```
1  Input: $\mathcal{R} = \{R_1, R_2, ..., R_n\}$, set of instance relations; $SG$,
   schema graph; $maxJoins$, max number of joins
2  Output: $\mathcal{L}$, lattice
3  // Generate the base level $\mathcal{L}_1$
4  $\mathcal{L}_1 \leftarrow \emptyset$
5  foreach $R_i \in \mathcal{R}$ do
6      for $1 \leq j \leq maxJoins + 1$ do
7          $\mathcal{L}_1 \leftarrow \mathcal{L}_1 \cup \{CreateSingleNodeGraph(R_i)\}$
8
9  // Generate higher levels $\mathcal{L}_k$, for $2 \leq k \leq maxJoins + 1$
10 foreach $2 \leq k \leq maxJoins + 1$ do
11     $\mathcal{L}_k \leftarrow \emptyset$
12     foreach Graph $G \in \mathcal{L}_{k-1}$ do
13         foreach $R \in Nodes(G)$ do
14             $\mathcal{G}' \leftarrow ExtendGraph(R, G, SG)$
15             foreach $G' \in \mathcal{G}'$ do
16                 // Offline Pruning 1: detect duplicates
17                 if $G' \notin \mathcal{L}_k$ then
18                     $\mathcal{L}_k \leftarrow \mathcal{L}_k \cup \{G'\}$
19 return $\mathcal{L}$
```

The details of the lattice generation algorithm are presented in Algorithm 1. It works as follows. We first create $maxJoins + 1$ copies for each input instance relation (lines 5 to 7). These constitute the bottom level of the lattice. We then construct the upper levels (lines 9 to 18). When generating the graphs at the level k (i.e., $\mathcal{L}_k$ for $2 \leq k \leq maxJoins + 1$), we check each graph G at the level $k-1$ (i.e., $\mathcal{L}_{k-1}$). For each relation R in G, we look up the schema graph SG to find possible edges that are connected with R. Whenever we find such an edge $e = (R, R')$, for each copy R'_c of R', we create a new graph G' by first copying G and then inserting the edge (R, R'_c) into G'. This is done by calling the function $ExtendGraph$ (line 14). For each such extension G', we then check whether $G' \in \mathcal{L}_k$. If not, G' is added into $\mathcal{L}_k$ (lines 15 to 18). Note that here, to detect the duplicates, we need to test

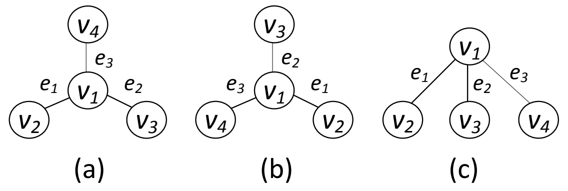


Figure 5: Two isomorphic trees and their canonical form.

the isomorphism between graphs, which is a problem not known as either P or NP-complete. However, since G' is a candidate join-query network, which by definition must be a tree [11], there are efficient algorithms (in linear time) for this special case. We use a variant of the algorithm in [3], by computing a *canonical labeling* for each graph (tree). Two graphs (trees) are isomorphic if and only if they have the same canonical labeling.

Specifically, given a candidate join-query network (tree) T, let $V(T)$ and $E(T)$ be its nodes and edges. For any $v \in V(T)$, the label of v is defined as the relation name R_i associated with v. For any $e \in E(T)$, the label of e is defined as $(R_i.a, S_j.b)$, where $R_i.a \bowtie S_j.b$ is the join associated with e. We further map the labels to integer ID's. Let $id(v)$ and $id(e)$ be the ID's assigned to a node v and an edge e. We compute the canonical labeling of T as shown in Algorithm 2. Example 3 illustrates this.

Algorithm 2: Canonical Labeling

```
1  Input: $T$, a candidate join-query network
2  Output: $l_T$, canonical labeling of $T$
3  GetCode($u$):
4  $l \leftarrow$"[$id(u)$"
5  if $HasChildren(u)$ then
6      $l.Append$("|")
7      foreach $v \in Children(u)$ do
8          $l(v) \leftarrow$"$id(e)GetCode(v)$" // $e = (u, v)$
9      Sort $v \in Children(u)$ with respect to $l(v)$
10     foreach $v \in Children(u)$ do
11         $l.Append(l(v))$
12 $l.Append$("]")
13 return $l$
14
15 Main:
16 $\mathcal{R} \leftarrow \{r | r = \arg\min_v\{id(v) | v \in V(T)\}\}$
17 $L_{\mathcal{R}} \leftarrow \{l_r | l_r \leftarrow getCode(r), r \in \mathcal{R}\}$
18 $l_T \leftarrow \min\{l_r | l_r \in L_{\mathcal{R}}\}$
19 return $l_T$
```

EXAMPLE 3 (CANONICAL LABELING). *Figure 5(a) and (b) show two isomorphic trees. Their canonical form is shown in (c). The corresponding canonical labeling computed by Algorithm 2 is:* $[v_1|e_1[v_2]e_2[v_3]e_3[v_4]]$.

In Algorithm 2, we first define $\mathcal{R}$ to be the set of nodes with the minimum node ID (line 16), and then call *GetCode* to compute the labeling l_r for each $r \in \mathcal{R}$ (line 17). The minimum l_r in lexicographic order is the canonical labeling for T (line 18). The procedure *GetCode* (lines 3 to 13) first puts the ID of the current node u into the labeling. If u has children, it appends a delimiter "|", and then recursively calls *GetNode* to construct the label $l(v)$ of each child node v (lines 7 to 8). The label of v is appended with respect to the ordering of $l(v)$ (lines 9 to 11).

Once the lattice is generated and duplicates are removed, each node is labeled with a SQL query corresponding to the node. Since Phase 0 is performed offline, the SQL query in each node has an uninstantiated "where" clause. More specifically, the join conditions (e.g., `Item`.cid = `Color`.id) in the "where" clause are present, but the keywords (e.g., `Color`.name contains "saffron" OR `Color`.synonym contains "saffron") can be added to it only at run-time, once user's keyword query is available. Keyword query is accepted in the next phase.

2.3 Keyword Based Pruning (Phase 1)

Once the user inputs the keyword query K, we *map* each keyword to a relation using an inverted index over the data. A keyword k_i can be *mapped to* a relation R if k_i occurs in some tuple in R. Recall that we generated copies $R_1 \ldots R_{m+1}$ for each relation R in the database. If k_i maps to R, k_i is bound to one of the copies R_j of relation R. Additionally, we bind the *empty keyword* to the copy R_0 for each relation R in the database.

We do this because relations to which no keywords are bound can still contribute to valid relationships. For example, for the keyword query "red candle", suppose that "red" is bound to the `Color` (C) table and "candle" is bound to the `Product Type` (P) table. While these two tables cannot be directly joined due to the lack of a key-foreign-key association, the `Items` (I) table can be used to form a path between them. The resulting join network is then $C_1 I_0 P_1$, which represents the query "Find all products where product type is *candle* and color is *red*". The I_0 here is used to indicate that no keyword is bound to the `Items` table. This is analogous to a *free tupleset* in Discover [11].

At the base level, all the nodes that contain queries with copies of relations to which no keyword is bound are pruned. Their respective ancestors are also pruned. For the keyword query "red candle", a sample lattice with the `Item` (I), `Color` (C), and `Product Type` (P) relations from the sample database in Figure 2 is presented in Figure 6. Upon using the inverted index "red" is bound to C_1 and "candle" to P_1. C_0, I_0 and P_0 are bound to the *empty keyword* and are not pruned. Only the shaded nodes in the lattice are retained. The remaining nodes are pruned.

In our implementation, we handle cases where keywords can have multiple interpretations by dealing with one interpretation at a time. Additionally, if a keyword does not occur anywhere in the database, the system displays all such keyword(s) and does not investigate the query any further. This is in accordance with "and" semantics for keyword search.

At the end of Phase 0, each node is labeled with a SQL query with an uninstantiated "where" clause. Once we bind keywords to copies of relations, the "where" clauses of the queries in each remaining node in the lattice can then be instantiated. We now have an instantiated, pruned lattice for the keyword query K.

2.4 Finding Answer and Non-Answer Query Nodes (Phase 2)

Once the lattice has been pruned based on keyword query K, the next step is to find nodes that contain queries that correspond to answer queries and non-answer queries. To do this, first we introduce some terminology.

- *Total/Partial Node*: A node can be total or partial. A node N is said to be total if its query contains tables corresponding to every keyword k_i in K. Otherwise N is said to be partial. Since we assume "and" semantics for keyword search, only a total node can contain an answer query. [2]

[2]We further note that totality decreases by moving down in the

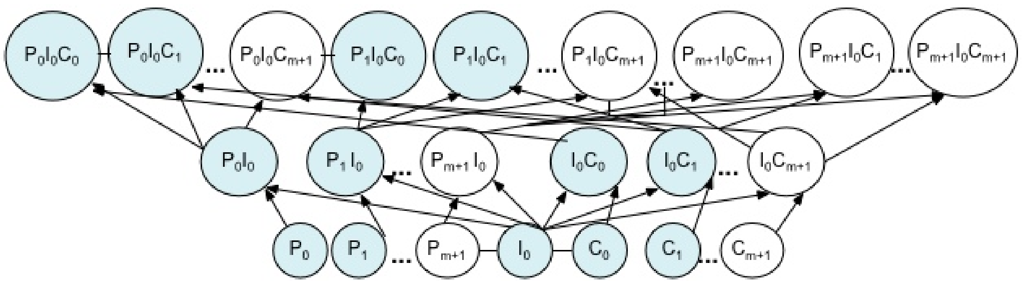


Figure 6: Sample lattice for the query "red candle". The un-shaded nodes are pruned.

- *Alive/Dead Node*: A node N is said to be alive if its query returns at least one tuple upon execution. If the query returns zero tuples, N is said to be dead. Typically a node can be classified as dead or alive only after executing its underlying structured query. As we shall see later in this section, in many cases, using the lattice structure helps us classify a node without actually executing its query.

- *Possibly Alive Node*: This node has not yet been classified as dead or alive. In the beginning, all the nodes in the pruned lattice are possibly alive.

- *Minimal-Total Node (MTN)*: A node N is said to be minimal-total, if N is total and no descendant of N is total. MTNs correspond to candidate networks in KWS-S systems [11], and contain answer and non-answer queries.

In Phase 2, we prune the lattice even further by only retaining MTNs and their descendants. To continue with our example, the node marked $P_1I_0C_1$ is the only MTN in the lattice in Figure 6. (None of the other shaded nodes are total.) We are now left with the task of classifying MTNs as dead or alive and explaining the reason(s) for the dead MTNs. We do this using Maximal Partial Alive Nodes (MPANs).

- *Maximal Partially Alive Node (MPAN)*: A node N is said to be a MPAN of a MTN M if it is both partial and alive, and if there exists no other node $N' \in \mathcal{D}esc(M)$ such that $N \in \mathcal{D}esc(N')$ and N' is alive.

There can be multiple reasons for a non-answer query. For example, the SQL query q_2 in Example 1 for the keyword query "saffron scented candle", where saffron is a scent, could be a non-answer query due to several reasons — the store carries products that are saffron scented but are not candles, they only carry unscented candles, they carry scented candles but none of them are saffron-scented or maybe they only carry products that are neither saffron-scented nor candles. The options that an administrator needs to explore in order to determine why q_2 is a non-answer query can get dauntingly large for manual debugging. Given that each keyword may have multiple interpretations (e.g., saffron could be a color or a scent), this task gets even more daunting.

We display the maximal alive query because we know that all its descendants are alive (i.e., if "find scented candles" returns some result tuples, then both "find scented products" and "find candles" will also return some result tuples). In Example 1, "find scented candles" and "find saffron scented products" are both MPANs of q_2. Collectively they convey that while the store does not carry saffron-scented candles, it does carry scented candles and other saffron-scented products. Notably, since all possible reasons for a non-answer query are sub-queries of the non-answer query, they can be systematically explored using our proposed lattice structure.

In the final phase of our system and in the rest of this section we discuss lattice traversal strategies to efficiently determine dead MTNs and their respective MPANs.

lattice, because the descendant sub-queries of a query q in general refer to fewer tables than q itself. This allows us to speak of *minimal* total nodes as in the following.

2.5 Lattice Traversal (Phase 3)

One approach to classifying the MTNs as dead or alive and finding MPANs is to simply execute the SQL queries for all the nodes in $\mathcal{L}$. However, this may not be necessary. Since each MTN is derived from its descendants, we can use the following two rules to avoid executing many of the SQL queries in $\mathcal{L}$.

Node Classification Rules:
- **(R1)** Node N is alive $\Rightarrow$ All $\mathcal{D}esc(N)$ are alive.
- **(R2)** If any $N' \in \mathcal{D}esc(N)$ is dead $\Rightarrow$ N is dead.

The descendants of a node in the lattice represent sub-queries of the node. R1 says that if a node is alive, all its sub-queries should also return some tuples when executed. R2 says that if a node is dead, all the queries of which it is a sub-query will also return no tuples. Next, utilizing the above two rules we propose traversal strategies that classify MTNs and find MPANs in the lattice.

2.5.1 Bottom-Up/Top-Down Traversal

In the bottom-up (BU) strategy, we classify one MTN at a time and traverse the sub-lattice consisting of the MTN and its descendants from the single-table level up. At each level we evaluate the SQL query corresponding to each node. If a node N is dead (i.e., its SQL query returns no result tuples), all the nodes in $\mathcal{A}sc(N)$, including the MTN, can be marked as dead (by R2). If a node is alive, it is marked as a potential MPAN until one of its ancestors is found to be alive. This process is repeated for all the MTNs until they are all classified as dead or alive and the corresponding MPANs are found.

This strategy performs well when the MTNs and MPANs corresponding to dead MTNs are found at lower levels of the lattice. In this case BU can avoid executing several expensive SQL queries at higher levels of the lattice. For keywords where MTNs and MPANs are found at higher levels, a better approach might be a top-down traversal of the lattice.

Top-down (TD) traversal is similar to BU, except that we traverse the sub-lattice for each MTN from its highest level down to the single-table level. We evaluate the SQL query corresponding to each node at each level. If a node N is alive, all the nodes in $\mathcal{D}esc(N)$ can be marked as alive (by R1). This is done till all the nodes in the lattice are classified and all the MPANs are found.

Algorithm 3: Bottom-Up with Reuse Approach

```
1  Input: $SG$, schema graph; $K = \{k_1, k_2, \ldots, k_N\}$, keyword
   query; $\mathcal{L}$, lattice; $M$, set of MTNs found during Phase 2
2  Output: $A$, set of alive MTNs; $D$, set of dead MTNs; $P$, set
   of corresponding MPANs for each dead MTN
3  GetBaseNodes($K$, $\mathcal{L}$):
4  $baseNodes \leftarrow \emptyset$, $nonKeywords \leftarrow \emptyset$
5  foreach $k \in K$ do
6      $T \leftarrow GetBaseTables(k)$
7      if $T == \emptyset$ then
8          $nonKeywords.add(k)$
9      else
10         $baseNodes.add(\mathcal{L}.GetBaseNodes(T))$
11     if $nonKeywords \neq \emptyset$ then
12         Display $nonKeywords$
13         $baseNodes \leftarrow \emptyset$
14 return $baseNodes$
15
16 Main:
17 $M' \leftarrow M$, $currLevel \leftarrow 1$
18 foreach $m \in M$ do
19     MP[$m$]$\leftarrow$ $GetDescendants(m)$ // potential MPANs
20 $B \leftarrow GetBaseNodes(K, \mathcal{L})$, $curr \leftarrow B$, $next \leftarrow \emptyset$
21 if $B == \emptyset$ then
22     return
23 while $currLevel \leq maxJoins + 1$ and $M' \neq \emptyset$ do
24     foreach $node \in curr$ do
25         $isAlive \leftarrow true$
26         if $node \notin B$ and $execSQL(node).nTuples == 0$ then
27             $isAlive \leftarrow false$
28         if $node \in M$ then
29             $M' \leftarrow M' - node$
30         if $isAlive == true$ and $node \notin M$ then
31             $next \leftarrow next \bigcup \mathcal{L}.GetParentNodes(node)$
32             foreach $m \in M$ do
33                 if $node \in$ MP[$m$] then
34                     $\mathcal{L}.RemoveAllDesc$(MP[$m$],$node$)
35         else if $isAlive == false$ then
36             $MarkAsDead(\mathcal{L}.GetAscNodes(node))$
37             if $node \notin M$ then
38                 foreach $m \in M$ do
39                     if $node \in$ MP[$m$] then
40                         $\mathcal{L}.RemoveAllAsc$(MP[$m$],$node$)
41                         $\mathcal{L}.Remove$(MP[$m$],$node$)
42             else
43                 $P[node]$$\leftarrow$ MP[$node$] // MPANs
44         $curr \leftarrow next$, $next \leftarrow \emptyset$
45     $currLevel \leftarrow currLevel + 1$
```

2.5.2 Bottom Up and Top Down with Reuse

We found that there is usually substantial overlap between the descendants of each MTN. Based on this observation, we modify BU and TD to process all the MTNs and their descendants simultaneously. We find that we can substantially reduce the redundancy in executing SQL queries corresponding to the common descendants of the MTNs. The corresponding algorithms are termed bottom-up with reuse (BUWR) and top-down with reuse (TDWR). The details of BUWR are presented in Algorithm 3. TDWR is very similar to BUWR so we do not elaborate on it any further.

Algorithm 3 works as follows. It first finds the descendants for the MTNs in M, which are all the potential MPANs (lines 18 to 19). It then traverses the lattice $\mathcal{L}$ in a bottom-up manner. For each keyword k, *GetBaseNodes* (lines 3 to 14) collects the base nodes (i.e., tables) in the lattice containing the keyword k. If some keyword is not contained by any base table, then this is reported to the user (lines 11 to 13) and no further exploration is needed (lines 21 to 22). Otherwise, answers and non-answers will be reported to the user by climbing up the lattice $\mathcal{L}$ (lines 23 to 45). For each node $node$ at the current level, the algorithm first checks its aliveness (if not known yet) by executing the SQL query associated with it (lines 25 to 27). If $node$ is alive, and it is not a MTN in M, then we can remove its descendants from candidate MPANs of each MTN $m \in M$, since they must be alive and cannot be MPANs because of the aliveness of $node$ (lines 30 to 34). On the other hand, if $node$ is dead, then all of its ancestors must be dead (line 36). If $node$ is not a MTN, once again we can remove $node$ and its ancestors from candidate MPANs of each MTN $m \in M$, since they cannot be MPANs because of their deadness (lines 37 to 41). Otherwise, $node$ is a dead MTN, and we need to report its MPANs (lines 42 to 43). Note that the MPANs must be those candidates that are still remaining in MP$[node]$. This is because, due to the nature of bottom-up traversal, the aliveness of each member in MP$[node]$ must have been either explicitly (by executing the corresponding SQL query) or implicitly (by using the two node classification rules R1 and R2) checked and hence known before $node$ is examined. After checking all the nodes at the current level, the algorithm can proceed to the next level (line 44). The next level only needs to include the parents of the alive nodes at the current level (line 31), since ancestors of dead nodes must also be dead and therefore can be excluded from further examination.

While we find that these approaches perform well in general, like any bottom-up or top-down approach, they suffer poor performance in certain cases. For example, TD will perform poorly if many MPANs are present at the lowest level in the lattice. On the contrary, BU will perform poorly if many MTNs at higher levels of the lattice are alive. In the rest of this section, we propose a score-based greedy approach, with the goal of avoiding the worst-case performance of both BU and TD.

2.5.3 Score Based Heuristic for Traversal

Given that the main advantage of the lattice structure is reuse amongst MTNs, the goal of this approach is to evaluate nodes in an opportunistic manner with the goal of minimizing the number of evaluated SQL queries. We do this by assigning a score to each unevaluated node in the lattice. This score indicates the amount of reduction in the search space that would result from evaluating this node. In other words, we evaluate the nodes that are most likely to influence the classification of other nodes first. Table 1 summarizes the notation used in the following discussion.

We start by investigating the effect that evaluating a node n_j in $\mathcal{L}$ has on the remaining nodes in the search space $\mathcal{S}(m_i)$ of each $m_i \in \mathcal{M}$. $\mathcal{S}(m_i)$ contains potential MPANs of m_i with unknown status. SQL queries might be required to determine aliveness of the nodes in $\mathcal{S}(m_i)$. Initially, $\mathcal{S}(m_i) = \mathcal{D}esc^+(m_i) = \{m_i\} \bigcup \mathcal{D}esc(m_i)$. Figure 7 demonstrates how n_j and m_i and their descendants could overlap. We next consider the cases when n_j is alive or dead.

- **If the current node n_j is alive** :

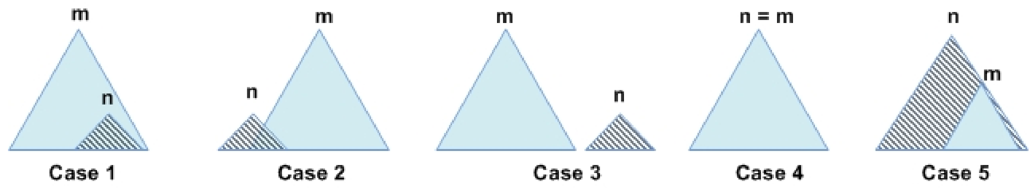


Figure 7: Ways in which a minimal complete node m, an unexplored node n, and their respective descendants may overlap.

Notation	Description
$\mathcal{N}$	$\mathcal{N} = \{n_1 \dots n_q\}$, the set of nodes in the lattice $\mathcal{L}$
$\mathcal{M}$	$\mathcal{M} = \{m_1 \dots m_p\}$, the set of MTNs
$\mathcal{P}(m_i)$	the set of MPANs for each dead $m_i \in \mathcal{M}$
$\mathcal{S}(m_i)$	the search space for each $m_i \in \mathcal{M}$ to find $\mathcal{P}(m_i)$
$\mathcal{D}esc(n)$	the set of descendants of the node n in $\mathcal{L}$
$\mathcal{A}sc(n)$	the set of ancestors of the node n in $\mathcal{L}$
$\mathcal{D}esc^+(n)$	$\mathcal{D}esc^+(n) = \{n\} \cup \mathcal{D}esc(n)$
$\mathcal{A}sc^+(n)$	$\mathcal{A}sc^+(n) = \{n\} \cup \mathcal{A}sc(n)$

Table 1: Notation used in the score-based heuristic

Case 1 ($n_j \in \mathcal{D}esc(m_i)$): If the current node n_j is a descendant of an MTN m_i, then all descendants of n_j are also alive. Since n_j is alive, these alive descendants cannot be maximal (i.e., cannot be in $\mathcal{P}(m_i)$), and thus can be removed from the search space $\mathcal{S}(m_i)$ as well.

$$\mathcal{S}(m_i) = \mathcal{S}(m_i) - \mathcal{D}esc^+(n_j).$$

Case 2 ($\mathcal{D}esc(n_j) \cap \mathcal{D}esc(m_i) \neq \emptyset$ and $n_j \notin \mathcal{D}esc(m_i)$): Let n_k be the root node of the intersection $\mathcal{D}esc(n_j) \cap \mathcal{D}esc(m_i)$. The descendants of n_k can also be removed from $\mathcal{S}(m_i)$ because they cannot be MPANs in $\mathcal{P}(m_i)$.

$$\mathcal{S}(m_i) = \mathcal{S}(m_i) - \mathcal{D}esc^+(n_k).$$

Case 3 ($\mathcal{D}esc(n_j) \cap \mathcal{D}esc(m_i) = \emptyset$): The intersection of the descendants of n_j and m_i is empty. This implies that n_j has no impact on the search space of m_i.

Case 4 ($n_j = m_i$): Here n_j is the MTN. $\mathcal{S}(m_i) = \emptyset$.

Case 5 ($m_i \in \mathcal{D}esc(n_j)$): This case cannot occur because it implies that m_i is not minimal and hence not an MTN.

• **If the current node n_j is dead** :

Case 1 ($n_j \in \mathcal{D}esc(m_i)$): In this case, all nodes in $\mathcal{A}sc(n_j)$ are also dead and therefore can be removed from $\mathcal{S}(m_i)$.

$$\mathcal{S}(m_i) = \mathcal{S}(m_i) - \mathcal{A}sc^+(n_j).$$

Case 2 ($\mathcal{D}esc(n_j) \cap \mathcal{D}esc(m_i) \neq \emptyset$ and $n_j \notin \mathcal{D}esc(m_i)$): No change in $\mathcal{S}(m_i)$.

Case 3 ($\mathcal{D}esc(n_j) \cap \mathcal{D}esc(m_i) = \emptyset$): No change in $\mathcal{S}(m_i)$.

Case 4 ($n_j = m_i$): m_i is the MTN.

$$\mathcal{S}(m_i) = \mathcal{S}(m_i) - \{n_j\}.$$

Case 5 ($m_i \in \mathcal{D}esc(n_j)$): This case cannot occur because it implies that m_i is not minimal and hence not an MTN.

We define $\mathcal{S}^a_{exp}(m_i)$ to be the expected search space of m_i if n_j is alive and $\mathcal{S}^d_{exp}(m_i)$ to be the expected search space of m_i if n_j is dead. Now, if p_a is the average probability that a node in the graph is alive, we define the score for n_j to be:

$$Score(n_j) = \sum_{m_i \in \mathcal{M}} p_a \cdot |\mathcal{S}^a_{exp}(m_i)| + (1 - p_a) \cdot |\mathcal{S}^d_{exp}(m_i)|. \quad (1)$$

Intuitively, this score measures the expected size of the search space given the information that n_j is alive/dead.

We give some further analysis to the score so defined. Let

$$\mathcal{C}over(n) = \{n\} \cup \mathcal{D}esc(n) \cup \mathcal{A}sc(n)$$

be the *coverage* of a node n. We can then express $\mathcal{S}^a_{exp}(m_i)$ and $\mathcal{S}^d_{exp}(m_i)$ more explicitly:

$$\begin{aligned} \mathcal{S}^a_{exp}(m_i) &= \mathcal{D}esc^+(m_i) - \mathcal{D}esc^+(n_j) \\ &= (\mathcal{D}esc^+(m_i) - \mathcal{C}over(n_j)) \\ &\cup\ (\mathcal{D}esc^+(m_i) \cap \mathcal{A}sc(n_j)) \end{aligned}$$

and

$$\begin{aligned} \mathcal{S}^d_{exp}(m_i) &= \mathcal{D}esc^+(m_i) - \mathcal{A}sc^+(n_j) \\ &= (\mathcal{D}esc^+(m_i) - \mathcal{C}over(n_j)) \\ &\cup\ (\mathcal{D}esc^+(m_i) \cap \mathcal{D}esc(n_j)). \end{aligned}$$

Since $\mathcal{D}esc^+(m_i) - \mathcal{C}over(n_j)$ and $\mathcal{D}esc^+(m_i) \cap \mathcal{A}sc(n_j)$ are disjoint, we have

$$\begin{aligned} |\mathcal{S}^a_{exp}(m_i)| &= |\mathcal{D}esc^+(m_i) - \mathcal{C}over(n_j)| \\ &+\ |\mathcal{D}esc^+(m_i) \cap \mathcal{A}sc(n_j)|, \end{aligned}$$

and similarly,

$$\begin{aligned} |\mathcal{S}^d_{exp}(m_i)| &= |\mathcal{D}esc^+(m_i) - \mathcal{C}over(n_j)| \\ &+\ |\mathcal{D}esc^+(m_i) \cap \mathcal{D}esc(n_j)|. \end{aligned}$$

Therefore, according to Equation (1), we have

$$\begin{aligned} Score(n_j) &= \sum_{m_i \in \mathcal{M}} |\mathcal{D}esc^+(m_i) - \mathcal{C}over(n_j)| \\ &+\ p_a \cdot \sum_{m_i \in \mathcal{M}} |\mathcal{D}esc^+(m_i) \cap \mathcal{A}sc(n_j)| \\ &+\ (1 - p_a) \cdot \sum_{m_i \in \mathcal{M}} |\mathcal{D}esc^+(m_i) \cap \mathcal{D}esc(n_j)|. \end{aligned}$$

Based on the above equation, we can see that the score actually takes three factors into consideration:

(1) *the descendants of the MTNs that are not covered by n_j (the 1st summand)*: these nodes must be explored no matter whether n_j is alive or dead;

(2) *the descendants of the MTNs that are ancestors of n_j (the 2nd summand)*: these nodes are explored when n_j is alive;

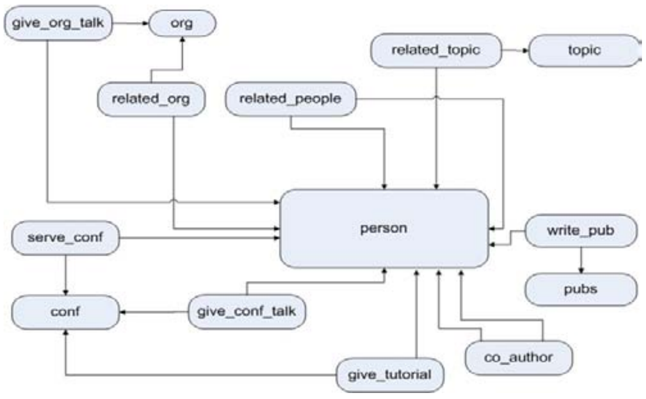


Figure 8: Relational schema for the DBLife dataset.

(3) *the descendants of the MTNs that are descendants of n_j (the 3rd summand)*: these nodes are explored when n_j is dead.

Hence, a smaller score means a smaller expected search space. We then use a simple greedy strategy based on this score to traverse the lattice — each time we pick the node n_j with the minimum score, check its aliveness, and eliminate other nodes from the lattice according to this information as before. This algorithm terminates when all nodes have been classified and the MPANs for all dead MTNs have been found.

The remaining issue is how to determine the probability p_a. We note here that setting p_a affects performance, not correctness. Estimating the probability value p_a accurately requires evaluating all the queries and finding out what percentage of them are empty. However, our experiments show that the simple assumption that p_a = 0.5 works surprisingly well. Nonetheless, it is still interesting future work to explore lightweight estimation approaches for p_a.

3. EVALUATION

In this section we evaluate our proposed approaches. We ran our experiments using PostgreSQL 8.3.6 on an Intel(R) Core(TM) 2 Duo 2.33 GHz system with 3GB of RAM. We implemented all the query processing algorithms in Java, and used JDBC to connect to the database. We further implemented the inverted indexes over the data using Lucene [18]. We evaluated the proposed approach over a 40MB snapshot of the DBLife [7] dataset that has 801,189 tuples in 14 tables (Figure 8). Note that in the DBLife schema, keywords are contained in 5 entity tables, namely, `Person`, `Publication`, `Conference`, `Organization`, and `Topic`. The 9 relationship tables connect the entity tables but do not contain any text attributes.

3.1 Lattice Generation

In Figure 9(a), we look at the number of nodes in the lattice. Having several copies of each table adds to the number of seed tables and consequently to the number of nodes in the lattice. As expected, the number of nodes grows exponentially as the level (and number of joins) increases. It is thus important to have efficient traversal and pruning strategies. Figure 9(a) shows the number of nodes generated and the number of duplicate nodes in the lattice. On average 11.7% of the generated nodes were removed due to duplicate elimination (note that the Y-axis is in log scale).

Next, in Figure 9(b), we look at the time spent in generating the lattice. We vary the level on the X-axis and measure the time taken on the Y-axis. We observe that lattice generation completes in less than 100 seconds, even for a lattice with 7 levels (i.e., 6 joins). We note that this is a one-time cost, and is done offline.

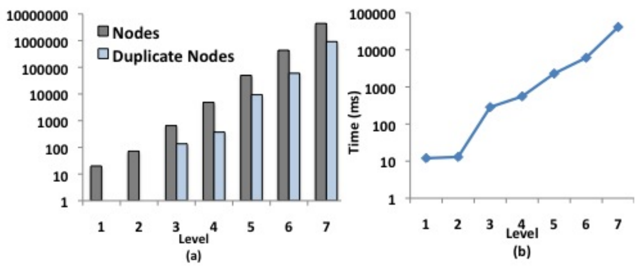


Figure 9: (a) The number of nodes generated in the lattice at each level and the number of duplicates are shown. As expected, the number of nodes in the lattice grows exponentially. On an average 11.7% of the nodes were pruned due to duplicate elimination. (b) The time spent in generating the lattice is shown. We note that the lattice is generated *offline*.

3.2 Keyword Queries

Table 2 lists the keyword queries we used in our following experiments. DBLife has a star schema, with the `Person` table at the center. As a result, queries with many person names (e.g., Q3) often lead to many MTNs. Q7, Q9, and Q10 do not contain any person names and Q8 contains the term "Washington" which occurs in the `Person`, `Publication`, and `Organization` tables. Q4 and Q6 lead to empty queries at the two-table level, but MTNs are found at higher levels as KWS-S explores relationships with more joins.

ID	Keyword Query
Q1	Widom Trio
Q2	Hristidis Keyword Search
Q3	Agrawal Chaudhuri Das
Q4	DeRose VLDB
Q5	Gray SIGMOD
Q6	DeWitt tutorial
Q7	Probabilistic Data
Q8	Probabilistic Data Washington
Q9	SIGMOD XML
Q10	Stream data histograms

Table 2: Keyword queries

3.3 Keyword Based Pruning (Phase 1) and Finding MTNs (Phase 2)

The next step involves mapping user's keyword query to schema terms and is performed online. This primarily involves lookups over the inverted indexes on the data. For the 10 testing queries, the time to map the keywords to schema terms varied between 7 ms and 66 ms with an average time of 26 ms.

Next, we measured the number of nodes in the lattice that remain in the lattice upon the introduction of keywords. We note that keyword-based pruning reduced the number of relevant nodes by 98% on average. Once the keywords are mapped to schema terms, the next step is to find the MTNs. This process took up to 23 ms for the 10 queries, in a lattice for level = 5. The number of MTNs ranged from 3 to 85.

Figure 10 summarizes these results. It also shows the number of unique descendants for the MTNs. We note that Q3 and Q8 have lower number of unique descendants, allowing higher possibility of reuse, as we will show later. We also computed these statistics for level = 7, and observed that the number of nodes after pruning varied from 277 to 18,904 with an average of 9,226 nodes, a reduction of 94.3% from the 161,440 nodes in the original lattice. The number of MTNs ranged from 35 to 1,418. This shows that even though a large number of lattice nodes are generated, phases 1 and 2 can prune the lattice substantially.

3.4 Comparison of Traversal Strategies

The goal of our traversal strategies is to efficiently determine if an MTN is alive or dead and to find the MPANs for the dead MTNs. We compared the five strategies for lattice traversal from end-to-end. Figure 11 shows the number of SQL queries that needed to be executed by each traversal approach for level = 5. Figure 12 shows the times taken by each approach for the corresponding queries. Note that both BUWR and TDWR perform better than their respective counterparts without reuse. This is especially true for Q3 and Q8, because for these queries, the total number of unique descendants are much smaller than that with duplicates. The number of SQL queries executed for Q3 is shown in Table 4. Q2 leads to only 3 MTNs, all of which are alive (i.e., Q2 has 0 MPANs). One of these 3 MTN queries is a join between the `Person` and `Publication` tables, with the keywords occurring in many tuples. This query takes around 20 seconds to execute. The proposed score-based heuristic (SBH) approach performs well in almost all cases, owing to its opportunistic choice of nodes during the pruning/traversal process. Further, we also note that TD and TDWR perform better than BU and BUWR respectively. Next, we explain the reason for the performance difference.

3.5 Impact of MTN and MPAN Distributions

For the DBLife dataset, we find that as the maximum level of the lattice increases, BU and BUWR generally perform poorly when compared to TD, TDWR, and SBH. This is because even keyword queries that return no answers at lower levels might return answers at higher levels. These answers correspond to relationships with many hops. Since the number of nodes in the lattice grows exponentially as the number of joins increases, TD and TDWR have a better pruning effect than BU and BUWR.

In Table 3, we present the distributions of MTNs and MPANs at levels 3, 5, and 7. Note that as most of the MTNs and MPANs are concentrated at higher levels, TD and TDWR perform better than BU and BUWR. We find that SBH performs well regardless of the distribution of MTNs.

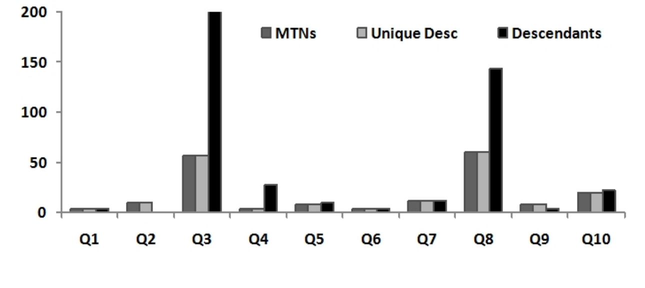


Figure 10: Keyword queries enable substantial pruning of the lattice (98% on an average). The number of MTNs and their descendants and unique descendants are presented to show the extent of overlap between the nodes in the lattice.

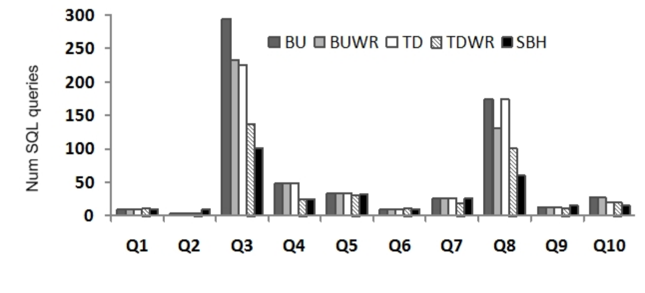


Figure 11: The number of SQL queries generated by the system for each keyword query.

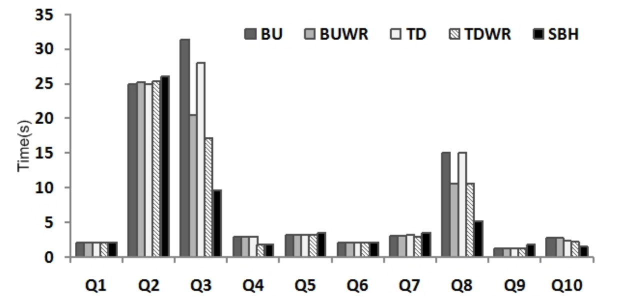


Figure 12: The time taken to execute all the SQL queries for each keyword query.

	MTNs			MPANs		
Q	L3	L5	L7	L3	L5	L7
Q1	1	6	41	0	4	34
Q2	0	3	35	0	0	0
Q3	0	85	1418	0	94	1584
Q4	4	20	144	8	28	130
Q5	4	24	164	2	10	42
Q6	1	6	41	2	4	18
Q7	2	14	92	4	12	70
Q8	0	31	451	0	87	1172
Q9	0	8	40	0	4	4
Q10	0	6	83	0	10	92

Table 3: Distributions of MTNs and MPANs at levels 3, 5, and 7 for the 10 keyword queries (L3 is short for "Level 3", and so forth). Note that most of the MTNs and MPANs are concentrated at higher levels.

3.6 Performance at Higher Levels

We now investigate how the approaches perform as we vary the maximum level of the lattice. Table 4 shows the number of SQL queries executed for Q3 when increasing the maximum level from 3 to 7. As before, we note that the number of SQL queries executed increases as the maximum level is increased. We also note that reuse-based approaches perform better by executing fewer queries — BUWR executes 28% fewer queries than BU, while TDWR executes 52% fewer queries than TD, at level 7. Further, TDWR performs better than BUWR, owing to the presence of a large number of MPANs and MTNs at higher levels of the lattice. Finally, we note that SBH provides substantial reduction (79% reduction compared to BU) in the number of queries executed at higher levels of the lattice.

3.7 Performance Improvement with Reuse

We were interested in investigating the extent of the overlap between the descendants of each MTN. Increased overlap would allow more reuse, decrease the number of SQL queries executed and consequently improve runtime performance. Figure 13 shows the percentage of reuse, i.e., $100 * (1 - \frac{N_u}{N})$, where N_u is the

Level	BU	BUWR	TD	TDWR	SBH
3	0	0	0	0	0
5	294	233	225	136	101
7	5036	3624	3866	1818	1026

Table 4: Number of SQL queries executed in all the traversal techniques for Q3 with lattice level = 3, 5, 7. SBH provides substantial reduction in the number of queries executed at higher levels of the lattice. The approaches with reuse perform better than their respective counterparts without reuse.

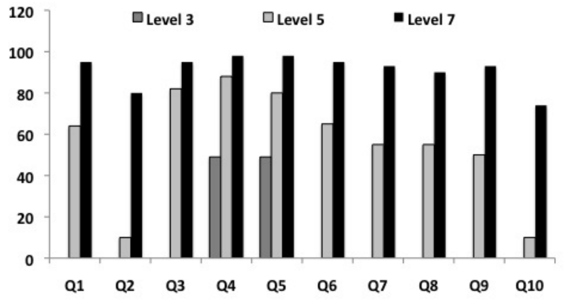


Figure 13: Percentage of reuse for the 10 keyword queries. While reuse is keyword dependent, it increases as the number of joins increases.

number of unique descendants of MTNs, and N is the total number of descendants of MTNs. The percentage of reuse for levels 3, 5, and 7 is shown. As expected, reuse increases as more joins are allowed. At level 3, only queries Q4 and Q5 show some overlap. However, we see substantial overlap between the descendants of the MTNs at levels 5 and 7. Specifically, Q2 and Q10 show a steep increase in overlap from level 5 to level 7. This increase is reflected in the performance of the reuse-based approaches and helps explain the performance of SBH over the other traversal approaches.

3.8 Comparison with Other Alternatives

Our proposed approach is not the unique one that can address the non-answer exploration problem. We therefore further compared our approach with other alternatives. Here we consider a simple indicator that is correlated with the "work" required to diagnose a non-answer. Our intent is to explore a simple quantitative metric that captures the intuition for why we think our approach may be helpful. Specifically, we compare our approach with the following two alternatives:

- *Return Nothing* (RN): Return nothing to the user for non-answers. This is the standard, existing KWS-S approach. To address the "why not" question, a developer would likely repeatedly submit modified queries by removing keywords from the original query. For instance, if the original keyword query were "k_1 k_2 k_3", then a user trying to understand the reason for the non-answer might additionally submit the queries: "k_1 k_2", "k_1 k_3", "k_2 k_3", "k_1", "k_2", and "k_3".
- *Return Everything* (RE): Do not build the lattice and return MPANs of the non-answers. Instead, explore alive descendants at runtime. For each descendant, issue the associated SQL query to determine its aliveness.

RN requires additional user effort to submit more queries. Both RE and our proposed approach remove this burden from the user. Meanwhile, the set of alive descendants returned by RN is both incomplete and redundant. It is incomplete, since only *minimal* alive descendants can be returned by existing KWS-S systems. That is, each leaf node of a candidate network is required to be bound by a keyword. As a result, alive descendants, including some MPANs, that do not satisfy this requirement will not appear in the results and hence are missing. It is also redundant, since some of the alive descendants returned may belong to answers (i.e., alive MTNs) but not non-answers (i.e., dead MTNs). Both incompleteness and redundancy are unsatisfactory for the purpose of debugging non-answers. On the other hand, RE returns the complete set of alive descendants. However, it is still redundant, since the aliveness of some descendants can be automatically inferred based on the aliveness of the others. Compared with RE, with the help of the lattice structure, our proposed approach can rule out this redundancy without sacrificing the completeness.

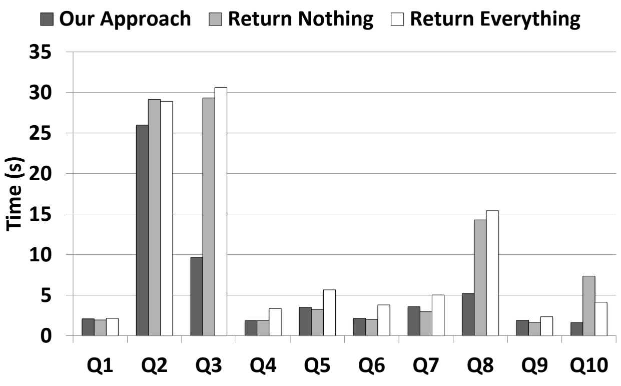


Figure 14: Response time when lattice level = 5

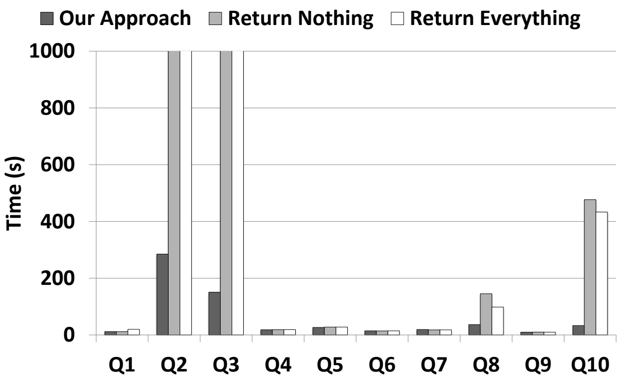


Figure 15: Response time when lattice level = 7

We further tested the system response time when the three approaches were adopted, in terms of the total execution time of the SQL queries issued. Figure 14 and 15 present the results for lattice levels 5 and 7. Our approach substantially reduces the response time for the more complicated queries Q2, Q3, Q8, and Q10, which contain three keywords, while the other queries only contain two. The improvement is more dramatic when multi-way joins (i.e., higher lattice levels) are allowed. For example, the response times are reduced by 84% and 99% for the two most costly queries Q2 and Q3, when the lattice level is 7 (i.e., up to 6 allowable joins).

4. RELATED WORK

Although we are not aware of any previous work that pertains to non-answers in the KWS-S context, the related work for the approaches and ideas presented in this paper is extensive.

Banks [1, 14], DBXplorer [2], and DISCOVER [11] are seminal papers in KWS-S. While Banks operates on a data graph, our paper is geared towards approaches like DISCOVER and DBXplorer, which use the underlying schema graph to explore relationships between the keywords. Several other KWS-S systems have been proposed over the years as well (see Yu et al. [25] for an extensive survey). Notably, Markowetz et al. [19] explored efficient generation and evaluation of candidate networks and briefly mentioned purging dead tuples in their paper about keyword search over streaming data. The Helix system [22] proposed a rule-based method for mapping keyword queries to structured queries. They automatically mined and manually tuned a set of patterns from query logs and mapped each pattern to a template query. A query was mapped to the best template once it arrived at runtime. Our static lattice structure is somewhat analogous to these templates. EASE [16] extensively leveraged offline computation to speed up runtime performance but did not consider the problem of non-answers. KWS-S-F [6] also leveraged offline computation but did not deal with non-answers. In addition, lattice structure has also been used in relaxing selection and join queries in relational databases to help users find more results [15].

We drew inspiration from Why Not [5] and Provenance of Non-Answers [12]. This work addressed non-answers in the context of single SQL queries and did not deal with KWS-S systems. Huang et al. [12] and Herschel et al. [8] provided tuple insertions or modifications that would yield the missing tuples. Chapman and Jagadish [5] tried to find the manipulation that led to a non-answer query. Tran and Chan [24] generated a modified query whose results included the user-specified missing tuple(s). In contrast, in a KWS-S system we cannot rely on user input, given that the user is assumed to be schema agnostic and may not even be aware that the KWS-S system is executing structured queries. Our notion of MPANs is similar to that of a frontier picky manipulation [5] in spirit. However, we feel that MPANs are more suited for the keyword search context; they represent the subset of keywords that would render a dead relationship alive. Work on lineage and provenance including [4, 20] influenced the lattice structure and use of MPANs to explain non-answer queries in KWS-S.

While we focused on pure relational database techniques in this paper, there has also been work on mapping sets of structured tuples to virtual documents and then applying information retrieval techniques to find results that match the keywords (e.g., the EKSO system [23]). However, this idea relies on inverted indices built over materialized views. In typical industry systems, these indices are updated only at some pre-determined time-intervals (mostly, on a nightly basis). This then implies that they may suffer from severe data staleness issues, for in the daily use of a relational database, any changes to the underlying data can impact answers and non-answers to keyword queries almost immediately. Moreover, it actually cannot work for non-answers if only "and" semantics is considered. Using "and" semantics, non-answers would never be displayed to the user. Nonetheless, this raises a very interesting point that there might be an alternative way to deal with the problem: replace "and" semantics by "or" semantics, though it seems to be equivalent to the "Returning Nothing" strategy discussed in Section 3.8 and thus suffers from issues such as incompleteness and redundancy. In fact, Hristidis et al. have considered the "or" semantics in KWS-S systems [10]. However, their focus was to efficiently present the end users with a list of top-k matches for moderate values of k. In contrast, our goal is to enable system developers to debug non-answers so providing top-k matches is insufficient. Nevertheless, this merits further exploration but it is out of the scope of this paper.

5. CONCLUSION

In this work we take a first step towards building a KWS-S system that exposes non-answer queries to system developers for the purpose of debugging the system. Given the exponential complexity of answer generation in KWS-S, exposing and explaining non-answer queries is a time-consuming process. We leveraged offline computation to generate a lattice structure and exploited the overlap between the queries to efficiently determine non-answer queries and their closest alive sub-queries.

While we concentrated on improving the performance of discovering and investigating non-answer queries in the KWS-S domain, this work opens up a couple of interesting directions for future work. For instance, debugging is often an interactive process and it is worth studying how to combine the search for MPANs with user intervention. Meanwhile, pushing user-defined constraints into the search procedure might greatly prune the search space and therefore significantly improve the efficiency. All of these are interesting questions that deserve further investigation.

Acknowledgements. We thank the anonymous reviewers for their valuable comments. This work was supported in part by NSF Grant III-1018792 and by a Google Focus Award.

6. REFERENCES

[1] B. Aditya, G. Bhalotia, S. Chakrabarti, A. Hulgeri, C. Nakhe, P. Parag, and S. Sudarshan. BANKS: browsing and keyword searching in relational databases. In *VLDB '02*.
[2] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: enabling keyword search over relational databases. In *SIGMOD '02*.
[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
[4] O. Benjelloun, A. D. Sarma, C. Hayworth, and J. Widom. An introduction to ULDBs and the Trio system. *IEEE Data Eng. Bull.*, 29(1):5–16, 2006.
[5] A. Chapman and H. V. Jagadish. Why Not? *SIGMOD '09*.
[6] E. Chu, A. Baid, X. Chai, A. Doan, and J. Naughton. Combining keyword search and forms for ad hoc querying of databases. In *SIGMOD '09*.
[7] DBLife. `http://dblife.cs.wisc.edu`.
[8] M. Herschel, M. A. Hernández, and W. C. Tan. Artemis: A system for analyzing missing answers. *PVLDB*, 2(2):1550–1553, 2009.
[9] HP Autonomy. `http://www.autonomy.com/`.
[10] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient ir-style keyword search over relational databases. In *VLDB*, pages 850–861, 2003.
[11] V. Hristidis and Y. Papakonstantinou. Discover: keyword search in relational databases. In *VLDB '02*.
[12] J. Huang, T. Chen, A. Doan, and J. Naughton. On the provenance of non-answers to queries over extracted data. *VLDB '08*.
[13] IBM Coremetrics. `http://www-01.ibm.com/software/marketing-solutions/coremetrics/`.
[14] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB '05*.
[15] N. Koudas, C. Li, A. K. H. Tung, and R. Vernica. Relaxing join and selection queries. In *VLDB*, pages 199–210, 2006.
[16] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou. EASE: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *SIGMOD '08*.
[17] F. Liu, C. Yu, W. Meng, and A. Chowdhury. Effective keyword search in relational databases. In *SIGMOD '06*.
[18] Lucene. `http://apache.lucene.org`.
[19] A. Markowetz, Y. Yang, and D. Papadias. Keyword search over relational tables and streams. *ACM Trans. Database Syst.*, 34(3).
[20] A. Meliou, W. Gatterbauer, K. F. Moore, and D. Suciu. Why so? or Why no? functional causality for explaining query answers. *CoRR*, abs/0912.5340, 2009.
[21] Orcale Endeca. `http://www.oracle.com/us/products/applications/commerce/endeca/`.
[22] S. Paparizos, A. Ntoulas, J. Shafer, and R. Agrawal. Answering web queries using structured data sources. In *SIGMOD '09*, pages 1127–1130, New York, NY, USA, 2009. ACM.
[23] Q. Su and J. Widom. Indexing relational database content offline for efficient keyword-based search. In *IDEAS*, pages 297–306, 2005.
[24] Q. T. Tran and C.-Y. Chan. How to ConQueR why-not questions. In *SIGMOD '10*, pages 15–26, New York, NY, USA, 2010. ACM.
[25] J. X. Yu, L. Qin, and L. Chang. *Keyword Search in Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010.

Elevating Annotation Summaries To First-Class Citizens In InsightNotes

Karim Ibrahim, Dongqing Xiao, Mohamed Eltabakh
Computer Science Department, Worcester Polytechnic Institute (WPI)
100 Institute Rd., Worcester, MA, USA
{kaibrahm, dxiao, meltabakh}@cs.wpi.edu

ABSTRACT

Most scientific and modern applications generate—in addition to the base data—valuable annotations and metadata information at unprecedented scale and complexity. Such annotations warrant the need for advanced annotation management techniques that not only propagate the raw annotations to end-users, but also mine, summarize, and extract useful knowledge from them. Towards this goal, we proposed the *InsightNotes* system, the first summary-based annotation management engine in relational databases [22]. InsightNotes relies on creating concise representations of the raw annotations, called *annotation summaries*. InsightNotes addresses several unique challenges related to the maintenance, propagation, and zooming of these summaries. However, a key limitation is that the annotation summaries are treated as *propagate-only* (*report-only*) objects that cannot be directly queried or manipulated. This limitation hinders higher-level applications from applying complex processing over both the base data and its attached annotation summaries even within a single query. In this paper, we propose new extensions to InsightNotes for treating the annotation summaries as *first-class citizens*. We address the challenges of: (1) Developing new manipulation functions and query operators specific for the annotation summaries, (2) Designing summary-based index structures and access methods for efficient retrieval and predicate evaluation, and (3) Extending the query optimizer to optimize queries accessing both the data and the annotation summaries. The proposed extensions not only make it feasible to natively query and manipulate the annotation summaries, but also achieve more than two orders of magnitude speedup in query evaluation.

1. INTRODUCTION

Metadata—usually referred to as *"annotations"*— is gaining an increasing importance in most modern database applications as a valuable source of information. Applications in many science domains, e.g., in biology, healthcare, earth sciences, and ornithology, create and manage annotations and metadata information in orders of magnitude larger than the base datasets as reported in [5, 20, 22]. For example, according to the *geneontology.org* website, several biological databases, e.g., Genobase (http://ecoli.naist.jp/GB8/), EcoliHouse (http://www.porteco.org/), and UniProt (http://www.ebi.ac.uk/uniprot), manage annotations in a 10x scale compared to the number of genes and proteins in the database. Moreover, in ornithological databases, e.g., DBRC (http://www.dbrc.org.uk/), and AKN (http://www.avianknowledge.net/), the number of annotations collected from the bird watchers and scientists all over the world is around 200x larger than the number of birds' collection stored in these repositories [1].

It is not only the scale of annotations that poses challenges, but also the need for transparent processing and propagation of annotations, and their combinatorial relationship with the data, e.g., annotations can be attached to single table cells (attributes), rows, columns, arbitrary sets and combinations of them, or even attached to sub-attributes. That is why annotation management has been extensively studied in RDBMSs to address some of these challenges [4, 7, 11, 14, 17, 21]. However, all of the existing techniques have the common limitation of manipulating only the raw annotations, and hence reporting back to end-users 100s of annotations attached to each output tuple. Nevertheless, any advanced processing of mining, summarizing, and extracting useful knowledge from the annotations is entirely delegated to end-users.

As a first step towards addressing the above limitations, we proposed the *"InsightNotes"* system, *a summary-based annotation management engine in relational databases* [22]. InsightNotes is based on integrating data mining and summarization techniques with annotation management in novel ways with the objective of creating concise and meaningful representations of the raw annotations, called *"annotation summaries"*. For example, the R.H.S in Figure 1 illustrates a data tuple with 100s of attached raw annotations, while the L.H.S illustrates the tuple with its attached summary objects using InsightNotes. The summary objects include, for example, Classifier-type objects, e.g., *ClassBird1* and *ClassBird2*, that classify the raw annotations into user-defined classes, Snippet-type objects, e.g., *TextSummary1*, that summarize the attached big articles and report snippets on each, and Cluster-type objects, e.g., *SimCluster*, that group similar annotations into groups and reports only a representative from each group. An overview on the system will be presented in Section 2.

1.1 Case Study: Effectiveness and Motivation

We performed a usability case study to demonstrate the effectiveness of InsightNotes and motivate the new extensions proposed in this paper. We used a small subset of 100 data tuples from the AKN ornithological database, each has a number of raw annotations ranging between 75 to 380. The annotations describe anything related to birds, e.g., color, body shape or weight, certain behavior or sound, eating habits, geographic location, or observed diseases. And then, we asked 20 students to query the data, and an-

10.5441/002/edbt.2015.06

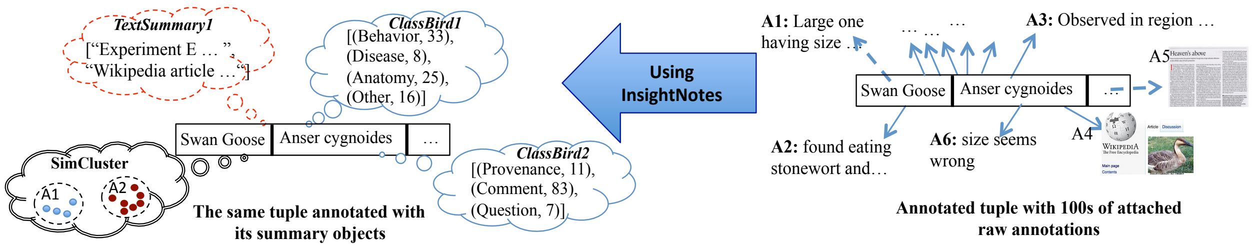

TextSummary1
["Experiment E … ", "Wikipedia article …"]
ClassBird1
[(Behavior, 33), (Disease, 8), (Anatomy, 25), (Other, 16)]
Swan Goose
Anser cygnoides
…
SimCluster
A1
A2
ClassBird2
[(Provenance, 11), (Comment, 83), (Question, 7)]
The same tuple annotated with its summary objects
Using InsightNotes
A1: Large one having size …
… … …
A3: Observed in region …
A5
Swan Goose
Anser cygnoides
…
A2: found eating stonewort and…
A6: size seems wrong
A4
Annotated tuple with 100s of attached raw annotations

Figure 1: Summary-Based Annotation Management in InsightNotes.

Query Semantics	# Qualifying data tuples	InsightNotes Group	Raw-Annotations Group
Q1: Report the *disease-related* annotations attached to birds with name like "Swan*".	5	Time: 47 sec Accuracy: 100%	Time: 21 mins False Positives: 17% False Negatives: 25%
Q2: Aggregate based on the bird's family column, and report the number of behavior-related information on each group.	3	Time: 47 sec Accuracy: 100%	Time: 45 mins False Positives: 18% False Negatives: 34%
Q3: Report the data tuples sorted based on the number of attached disease-related annotations	100	Time: 5.2 mins Accuracy: 100%	-----

Figure 2: Usability Case Study using InsightNotes.

swer the three questions highlighted in Figure 2. These are simple annotation-based analytical queries that scientists or end-users may ask over their datasets. Half of the students use the InsightNotes engine, while the other half uses an existing annotation management engine that reports the raw annotations [11]. We then measured the average time taken by each group (including writing the query) as well as the results' accuracy.

To answer $Q1$, the InsightNotes group needs to submit a single SQL query to get the 5 expected data tuples (similar to the L.H.S in Figure 1). And then, they need to issue another follow-up command, i.e., a zoom-in command, to retrieve the raw disease-related annotations over these tuples. In contrast, the *Raw-Annotations* group will get the 5 tuples along with their raw annotations (similar to the R.H.S in Figure 1). And then, they need to manually read the annotations and extract the desired ones. It took them, on average, 21 minutes and they reported the results with high false-positive and false-negative ratios as indicated in the figure.

To answer $Q2$, the InsightNotes group needs to only retrieve the number of the behavior-related annotations from the answer, i.e., *ClassBird1.Behavior*. It took them few seconds for writing and executing the query. In contract, the other group took very long time and still produced erroneous results—Notice that $Q2$ is an aggregation query, and thus each output tuple may have many annotations collected from multiple base tuples.

The $Q3$ query is more challenging because InsightNotes does not provide mechanisms for sorting the data based on their attached summaries. Thus, the InsightNotes group needed to go over the 100 reported tuples, and manually sort them according to the *ClassBird1.Disease* field. For the other group, it was not even feasible to analyze 100s of annotations over each of the reported tuples to figure out the number of disease-related annotations, and then sort based on that.

1.2 Limitations and Proposed Extensions

The results from the our study show that InsightNotes opens a promising direction for better understanding of large-scale annotations and extracting useful knowledge from them. However, the results also show that InsightNotes has the critical limitation of treating the annotation summaries as *"propagate-only objects"*. This limitation hinders the applications from mixing operations over both the data content and annotation summaries even within a single query (Refer to $Q3$ in Figure 2). In this paper, we propose extending the InsightNotes system by elevating the annotation summaries to be first-class citizens, where end-users and applications can manipulate them in various ways, e.g., selecting, joining, or ordering the data tuples based on their attached annotation summaries. To build such full-fledged summary-based annotation management engine, we propose the following contributions:

- **Seamless Manipulation of Diverse Summary Types:** InsightNotes supports three types of summarization techniques, i.e., clustering, classification, and text summarization. And hence, the summary objects attached to the data tuples can have diverse types, structures, and properties (Refer to Figure 1). Therefore, we propose manipulation functions at different granularities, e.g., at the *tuple-level* to manipulate the entire set of attached summary objects, and at the *object-level* to manipulate the individual summary objects according to their types.

- **Summary-Based Query Processing:** We propose building an extended query engine, where end-users can process both the data and their attached annotation summaries seamlessly in a single query plan. For example, $Q3$ in Figure 2 involves a *summary-based ordering* operation. Another query may be interested in retrieving only the data tuples with zero provenance-related annotations, i.e., `ClassBird2.Provenance = 0`, which involves a *summary-based selection* operation. Therefore, we extend the InsightNotes's query engine by introducing new summary-based query operators, e.g., filter, selection, join, and sort, that operate on the summaries' content. We define their algebraic semantics and integrate them with the standard operators in a single query plan.

- **Efficient Access Methods and Retrieval Mechanisms:** Applying predicates and operators on top of the annotation summaries warrants the need for efficient retrieval mechanisms and indexing techniques to achieve scalable performance. For example, how could the system efficiently answer the two queries mentioned above, i.e., a selection query based on `ClassBird2.Provenance = 0`, and an ordering query based on `ClassBird1.Disease`. We propose summary-based indexing techniques that achieve efficient execution for the summary-based queries, while retaining the optimal summary-propagation performance.

- **Extended Summary-Based Query Optimizer:** The integration between the summary-based and the standard SQL operators within a single query engine opens several new opportunities for query optimization. For example, one query may now involve joins and selections based on both the data and the summaries, and thus the query optimization becomes even more challenging. Therefore, we introduce several new equivalence and transformation rules as well as an extended cost model that guide the query optimizer in generating efficient execution plans.

• **Realization and Evaluation:** We developed the proposed extensions within the InsightNotes prototype engine [22]. The experimental analysis demonstrates the value-added functionalities of directly manipulating and querying the annotation summaries, e.g., enabling a seamless expression of more complex annotation-based analytical queries, and the significant performance gain from the proposed optimizations.

The rest of the paper is organized as follows. In Section 2, we overview the InsightNotes system. In Section 3, we present the new summary-based functions and query operators. Sections 4 and 5 introduce the summary-based indexing scheme, and the extended query optimizer, respectively. The experimental evaluation is presented in Section 6, while the related work is presented in Section 7. Finally, the conclusion remarks are included in Section 8.

2. OVERVIEW ON InsightNotes SYSTEM

InsightNotes addresses several challenges related to managing the annotation summaries, which include: (1) Designing an extensible engine where domain experts and database admins can define how to summarize and mine their annotations, (2) Developing efficient and incremental mechanisms for the maintenance of annotation summaries to scale up with large number of annotations, (3) Extending the query engine and relational algebra to operate on and propagate the annotation summaries along with the queries' answers, and (4) Building zoom-in query processing mechanisms that enable end-users to zoom-in and retrieve the raw annotations of specific summaries of interest. In this section, we overview the basic functionalities of InsightNotes needed for this paper.

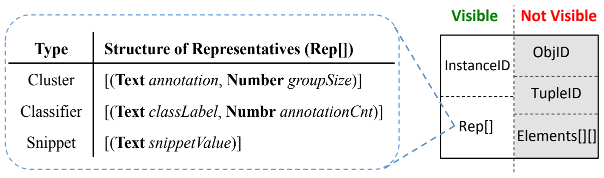
2.1 InsightNotes's Data Model

The system supports three widely-used families (types) of mining and summarization techniques, which are: *Text Summarization*, *Clustering*, and *Classification* techniques. The system is extensible such that the database admins can customize these techniques—and instantiate what is called *Summary Instances*—to fit their domains and produce the desired summaries. Each user relation R can be linked to as many summary instances as needed. For example, Figure 1 illustrates Table Birds having four summary instances linked to it (2 Classifiers, 1 Snippet, and 1 Cluster). Therefore, the raw annotations attached to each data tuple in this table (the R.H.S) will be summarized according to these four summary instances. This will result in creating the *Summary Objects*, which will be attached back to the corresponding data tuple (the L.H.S).

Assume a user's relation R having n data attributes and k summary instances linked to it. Then, each tuple $r \in R$ has the following conceptual schema:

$$r =< a_1, a_2, ..., a_n, \{s_1, s_2, ..., s_k\} >$$

where $a_1, a_2, ..., a_n$ are the data values of r, and $s_1, s_2, ..., s_k$ are the summary objects attached to r. Each summary object consists of a five-ary vector {ObjID, InstanceID, TupleID, Rep[], Elements[][]} as depicted in the following figure:



Type	Structure of Representatives (Rep[])
Cluster	[(**Text** *annotation*, **Number** *groupSize*)]
Classifier	[(**Text** *classLabel*, **Numbr** *annotationCnt*)]
Snippet	[(**Text** *snippetValue*)]

The *ObjID* is the objects's unique identifier, and the *InstanceID* and *TupleID* are references for the corresponding summary instance, and the data tuple, respectively. The *Rep[]* array stores the representatives produced from the summarization algorithm, while *Elements[][]* is a two-dimensional array storing for each representative, the references (Ids) to its contributing raw annotations. At query time, end-users will see only the *InstanceID* and *Rep[]* fields of each propagated summary object as illustrated in Figure 1.

For each summary object s_i, the structure of its representatives stored in Rep[] depends on s_i's type as depicted in the above figure. For example, in the case of the *Cluster* type, each cluster (group) will report an annotation as its representative as well as the number of annotations in that group. Hence, the Rep[] array consists of a list of representatives in the form of pairs [(*Text annotation, Number groupSize*)]. In the case of the *Classifier* type, each representative will have a class label along with the number of annotations assigned to this label. For the *Snippet* type, each large annotation will have a corresponding short snippet as its representative.

2.2 Summary-Aware Query Processing and Propagation

InsightNotes's query engine has several extensions that enable efficient and seamless propagation of the summary objects under complex transformations, e.g., projection, join, grouping and aggregations, and duplicate elimination. We proposed extensions to the semantics and algebra of each query operator to manipulate the summary objects on-the-fly without the need for accessing the raw annotations. The following example demonstrates a Select-Project-Join (SPJ) query involving summary propagation in InsightNotes. The formal semantics of all query operators can be found in [22].

Example 1: *Assume an SQL query* `"Select r.a, r.b, s.z From R r, S s Where r.a = s.x And r.b = 2"` *over the two tuples r and s presented in Figure 3. Tuple r has four summary objects attached to it, while tuple s has only two attached summary objects. We proved in [22] in Theorems 1 and 2 that to guarantee identical summary propagation under different—but equivalent—query plans, InsightNotes needs to project out the un-needed annotations before any merge operation over the summary objects. Therefore, the projection operator in Step 1 in Figure 3 projects out attributes $r.c$ and $r.d$ and eliminates the effect of their annotations from r's summary objects. For example, the* `annotationCnt` *field in the classifier objects is decremented, the wikipedia article in the snippet object is deleted, and the cluster objects are modified, e.g., some annotations are dropped from each cluster, and hence the* `groupSize` *field is decremented. Moreover, if a cluster's representative is dropped, then another representative is elected (See $A5$ representative replacing the dropped $A2$ representative). The same operation takes place over tuple s, where the effect of all annotations attached to both $s.x$ and $s.y$ is removed from s's summary objects. The only difference is that $s.x$ attribute will not be projected out because it is needed in the subsequent join operator.*

The next operator in the query plan is the selection operator over r (Step 2). Based on the query's predicate, r will pass the operator and all its summary objects will propagate without any change. Then, the produced tuples will join and their summary objects will be merged (Step 3). According to the merge procedure, r's summary objects `ClassBird1` *and* `TextSummary1` *will propagate without any change since they do have no counterpart objects over s. Whereas summary objects* `ClassBird2` *and* `SimCluster` *will be combined. This action takes into account the case where the same annotation may be attached to both tuples r and s, and hence the annotation's effect on the summary objects should not be double counted. For example, assuming that there are five common annotations on both r and s classified as* `Comment`*, then when the two objects are merged the sum of that classifier label will be 22 instead of 27 as illustrated in the figure. The merge of the clus-*

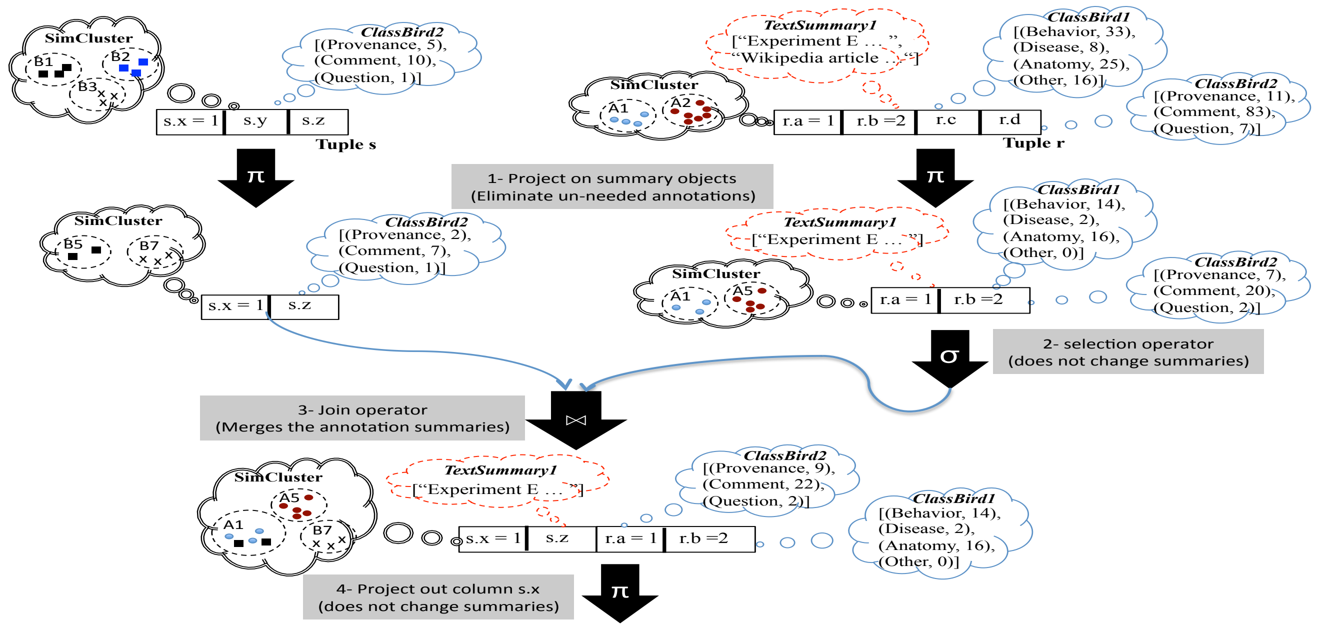

SimCluster
B1
B2
B3
ClassBird2
[(Provenance, 5), (Comment, 10), (Question, 1)]
s.x = 1
s.y
s.z
Tuple s
TextSummary1
["Experiment E … ", "Wikipedia article … "]
ClassBird1
[(Behavior, 33), (Disease, 8), (Anatomy, 25), (Other, 16)]
SimCluster
A1
A2
r.a = 1
r.b =2
r.c
r.d
ClassBird2
[(Provenance, 11), (Comment, 83), (Question, 7)]
Tuple r
π
1- Project on summary objects
(Eliminate un-needed annotations)
π
SimCluster
B5
B7
ClassBird2
[(Provenance, 2), (Comment, 7), (Question, 1)]
s.x = 1
s.z
TextSummary1
["Experiment E … "]
ClassBird1
[(Behavior, 14), (Disease, 2), (Anatomy, 16), (Other, 0)]
SimCluster
A1
A5
r.a = 1
r.b =2
ClassBird2
[(Provenance, 7), (Comment, 20), (Question, 2)]
σ
2- selection operator
(does not change summaries)
3- Join operator
(Merges the annotation summaries)
⋈
SimCluster
A5
A1
B7
TextSummary1
["Experiment E … "]
ClassBird2
[(Provenance, 9), (Comment, 22), (Question, 2)]
s.x = 1
s.z
r.a = 1
r.b =2
ClassBird1
[(Behavior, 14), (Disease, 2), (Anatomy, 16), (Other, 0)]
4- Project out column s.x
(does not change summaries)
π

Figure 3: Example Query in InsightNotes.

ter summary objects is slightly more complex. The main idea is that the overlapping groups from both sides, e.g., the groups represented by $A1$ and $B5$, will be combined together, whereas the non-overlapping groups, e.g., the groups represented by $A5$ and $B7$, will propagate separately as illustrated in the figure. Finally attribute $s.x$ will be projected out before producing the output.

3. SUMMARY-BASED FUNCTIONS & OPERATORS

3.1 Summary-Based Manipulation Functions

The first step in treating the annotation summaries as first-class citizens is to design a set of interfaces and manipulation functions on top of them. In the following, we demonstrate few of the developed functions, which we use throughout the paper. We also expect the end-users to leverage these basic functions to create more semantic-rich summary-based UDFs.

• **Summary Set Functions:** We introduce a special variable "\$" for each data tuple that represents the set of summary objects attached to this tuple, i.e., $r.\$$ represents the set of summary objects attached to r. Then, we define interface functions over the \$ variable, which include:

∘ **Int \$.getSize():** Returns the number of summary objects within the set. For example, referring to tuple r in Table `Birds` in Figure 1(c), $r.\$.getSize() = 4$.

∘ **SummaryObj \$.getSummaryObject(String** *InstName***):** The function takes a summary instance name as input, and returns the summary object corresponding to that name, otherwise it returns Null. For example, $r.\$.getSummaryObject(‘ClassBird1')$ and $r.\$.getSummaryObject(‘TextSummary1')$ return the Classifier and Snippet summary objects attached to tuple r.

∘ **SummaryObj \$.getSummaryObject(Int** i **):** This function takes a position within the summary set as input, and returns the summary object at that position. Since the objects in the set do not follow a pre-defined order, this function is more useful when used within UDFs, e.g., to iterate over the objects within a summary set and apply a certain functionality.

We then define a set of manipulation functions over each summary object O according to its summary type. Some functions are common to all types. For example, O**.getSummaryType()** and O**.getSummaryName()**, return the type of the summary object—as either "Classifier", "Snippet", or "Cluster"—, and the summary instance name, respectively. Another common function to all types is O**.getSize()**, which returns the number of representatives within object O, i.e., the size of $O.Rep[]$. For example, referring to Figure 1, the *ClassBird1* classifier object has 4 representatives, while *SimCluster* cluster object has 2 representatives. Other functions are specific to each summary type. For example:

• **Classifier Type Functions:** For a summary object O of type Classifier, the defined functions include:

∘ **String** O**.getLabelName(Int** i**):** Returns the class label at position i, i.e., Rep[i].classLabel. The order among the class labels is pre-defined based on the order specified when creating the classifier summary instance in the system.

∘ **Int** O**.getLabelValue(Int** i | **String** *label***):** This function takes either an index i or a class label *label* as input, and returns the corresponding value, i.e., Rep[i].annotationCnt (for input i), or Rep[j].annotationCnt, where Rep[j].classLabel = *label* (for input *label*).

• **Snippet Type Functions:** For a summary object O of type Snippet, the defined functions include:

∘ **String** O**.getSnippet(Int** i**):** Returns the snippet value at position i. The order among the snippets is arbitrary and does not follow a pre-defined order.

∘ **Boolean** O**.containsSingle(String** kw_1 **[, String** kw_2**, ...]):** Returns True if all of the given keywords $kw_1, kw_2, \ldots$ are contained within any one of O's snippets or the raw annotations. As we studied in [16], there is a tradeoff—w.r.t accuracy and performance—between searching the snippets vs. searching the raw annotations.

○ **Boolean O.containsUnion(String kw_1 [, String kw_2, ...]):** Returns True if all of the given keywords $kw_1, kw_2, ...$ are contained within the union of O's snippets or O's raw annotations. In this function, the keywords may span multiple annotations attached to the same tuple.

Internally, InsightNotes—which uses PostgreSQL as its underlying DBMS— implements the summary objects as composite data types. On top of these types, the manipulation functions presented above are defined.

3.2 Summary-Based Relational Operators

We now introduce several summary-based relational operators. Unlike the standard SQL operators, these operators operate on the summary objects attached to each tuple instead of its data content. The summary-based operators can be mixed with other standard relational operators in a single query pipeline for seamless processing. The new operators include:

• **Filter Operator ($\mathcal{F}_p(R)$):** The filter operator takes a set of summary-based predicates p, and returns each tuple $r \in R$ along with only its summary objects satisfying p. The operator is formally defined as:

$$\mathcal{F}_p(r) = \{r' =< a_1, a_2, ..., a_n, \{s_i, ...\} > \mid p(s_i) = True, \text{ where } 1 \leq \text{i} \leq \text{k} \}$$

For example, referring to Figure 1(c), the predicate `(getSummaryName() = 'SimCluster')` returns r along with only the the specified cluster summary object. In contrast, the predicates `(getSummaryType() = 'Classifier')` return r along with only the two classifier summary objects *ClassBird1* and *ClassBird2*.

• **Selection Operator ($\mathcal{S}_p(R)$):** The summary-based selection operator takes a set of summary-based predicates p, and returns the data tuples $r \in R$ having summary objects satisfying p. Otherwise, r is dropped. For qualifying tuples, all their summary objects will pass without change. The algebraic expression of the operator is as follows:

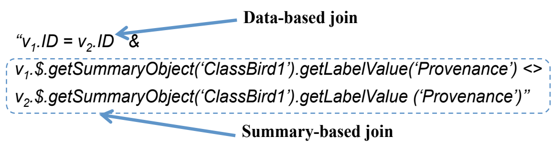
$$\mathcal{S}_p(R) = \{r \in R,\ r =< a_1, a_2, ..., a_n, \{s_1, s_2, ..., s_k\} > \mid p(r.\$) = True\}$$

The summary-based predicates may range from black-box UDFs that take $r.\$$ as a parameter and return a Boolean value, to explicit predicates based on the system-defined manipulation functions presented in Section 3.1. In the latter case, the system can reason about and optimize the execution of these predicates as will be presented in Section 4. For example, the predicate `(r.$.getSummaryObject('ClassBird2').getLabelValue('Provenance') = 0)` returns only R's tuples having no provenance-related annotations attached to them. In contrast, the predicate `(r.$.getSummaryObject('TextSummary1').containsSingle('Wikipedia', 'hormone'))` returns R's tuples that have at least one annotation containing both keywords. Such predicates can be efficiently evaluated using the the summary-based indexes presented in Section 4.

• **Join Operator ($\mathcal{J}_p(R, S)$):** The summary-based join operator joins two input tuples $r \in R$ and $s \in S$ iff the summary-based join predicates p evaluate to True over $r.\$$ and $s.\$$. The algebraic expression of the operator is as follows:

$$\mathcal{J}_p(R, S) = \{< r, s >, where\ r \in R\ \&\ s \in S \mid p(r.\$, s.\$) = True\}$$

For example, referring to Table *Birds* in Figure 1, assume we have two revisions of this table, V_1 (after Revision 1) and V_2 (after Revision 2). Then, reporting the data tuples whose number of provenance annotations has changed between the two revisions would involve the following expression:



"v_1.ID = v_2.ID &
v_1.$.getSummaryObject('ClassBird1').getLabelValue('Provenance') <>
v_2.$.getSummaryObject('ClassBird1').getLabelValue ('Provenance')"

The expression combines both data- and summary-based join operators. As will be discussed in Section 5 and based on the available indexes and statistics, the query optimizer may decide to join the tuples based on the data values and then applies a summary-based selection operator, i.e., ($\mathcal{S}(R \bowtie S)$). Alternatively, it may join the tuples based on the summary objects and then applies a standard selection operator ($\rho(\mathcal{J}(R, S))$).

• **Sort Operator ($\mathcal{O}_{f[,direction]}(R)$):** The summary-based sort operator orders the data tuples in R according to the summary-based function $f(r.\$)$. Function f must return values of a data type having a *full-ordering* property, e.g., number, string, and Boolean. Using the $\mathcal{O}$ sort operator, the $Q3$ query in the case study (Figure 2) can now be fully automated and answered in few seconds.

It is worth highlighting that these summary-based operators are new physical operators introduced to the InsightNotes engine. They are not implemented as UDFs within PostgreSQL DBMSs for the following fundamental reasons: (1) If the summary-based operators are implemented as UDFs, then their execution will be carried out and encapsulated within the standard SQL operators. As a result, none of the summary-based optimizations proposed in Section 5 would have been possible. (2) The annotation summaries are not like any other user-defined data types created through PostgreSQL extensibility. They are special tuple-based metadata information that requires extending the semantics of the core query operators [22]. That is why the core operators of InsightNotes in [22] do not manipulate the summaries through UDFs, and consequently, the newly proposed operators cannot be implemented as UDFs. And (3) The design choice of implementing the summary-based operators as new physical operators does not limit the extensibility of InsightNotes because the operators are defined at the summary-type level, i.e., Classifier, Snippet, and Cluster types. And thus, they apply to any driven instance under those types.

4. SUMMARY-BASED INDEX SCHEME

To enable efficient execution of the summary-based relational operators, we need to build a summary-based indexing scheme over the summary objects. In this paper, we will focus only on the Classifier-Type indexing scheme. The InsightNotes system will not automatically index all summary instances defined in the database. Instead, this process is triggered by DB admins using the following command:

```
Alter Table <tableName>
  [Add [Indexable] | Drop] <InstanceName>;
```

This extended SQL command is used in InsightNotes to link a Summary Instance SI to a given user's relation R (Refer to Section 2.1). The newly added optional clause `Indexable` will inform the system to build an index on SI's summary objects created over R's tuples.

Before we investigate possible indexing scheme, we briefly explain how the summary objects are currently stored in InsightNotes to optimize their propagation at query time. Referring to Figure 4(a), given a user's relation R, each tuple in R may have one or more summary objects attached to it according to number of summary instances linked to R. To optimize the propagation of the annotation summaries at query time, R's summary objects are stored in a de-normalized form in a corresponding catalog table *R_SummaryStorage*, as illustrated in Figure 4(b). Each tuple in R has a corresponding unique tuple in *R_SummaryStorage* linked together through unique tuple identifiers (OIDs). This scheme has two main advantages: (1) Since the summary objects are stored in tables separate from the data tables, there is no I/O or CPU overheads added to users' relations when queried in isolation, i.e., when the data is queried without annotation propagation, and (2) Since the summary objects are stored in a de-normalized form, there is no additional I/O or CPU overheads at query time to re-construct them from their primitive components. Thus, their propagations becomes more efficient as studied in [22].

4.1 Classifier-Type Indexing Scheme

Target Query: *The index will speedup summary-based selection operators in the form of* `"classLabel <Op> constant"`, *where* `classLabel` *is a classifier label within a classifier summary object, and Op is a comparison operator including* $\{=, >, <, \leq, \geq\}$. *The output are the data tuples whose classifier summary objects satisfy the given predicates. The index will also speedup summary-based join and sorting operators involving the indexed classifier column.*

Example 2: *Referring to Figure 4(a), assume we want to retrieve the data tuples having more than 5 associated questions. The SQL query will be:*

```
Select * From R r
   Where r.$.getSummaryObject('ClassBird2').
   getLabelValue('Question') > 5;
```

Baseline Indexing Scheme: A straightforward indexing strategy over the Classifier-type summary objects is to normalize their representation by replicating their components, i.e., the class labels and their counts, and storing them in a separate table (See Figure 4(c)). And then, we can build a standard B-Tree index on each of the columns, i.e., the `ClassLabel` and `Cnt` columns. Moreover, since most predicates over the Classifier-type objects will reference both columns, we may create a third system-maintained (derived) column that concatenates these two columns, and then index its values using the B-Tree index as illustrated in Figure 4(c).

The advantage of this scheme is that it uses the standard indexes without modifications. However, it has two major drawbacks. First, the storage overhead of the summary objects is doubled; one replicate is for efficient propagation, and another replica is for indexing. And second, starting from the index to reach the actual data tuples in relation R, we will need several join operations among multiple tables, which certainly degrades the query performance. The proposed Summary-BTree indexing scheme will overcome these limitations.

4.1.1 Summary-BTree Index Structure

The proposed *Summary-BTree* index is a variant of the standard B-Tree that can be directly built over the de-normalized representation of the Classifier-type summary objects. The structure of the index is depicted in Figure 4(d). Assume the index is built on top of the summary instance *ClassBird1* defined on Relation R. The creation of the index involves three steps:

○ **Itemization:** The Rep[] array within the object will be *itemized* by converting the array elements *(String classLabel, Integer AnnotationCnt)* to a sequence of text values in the form of `"classLable:ExtendedAnnotationCnt"` as illustrated in Figure 4(c) Step 1. The *ExtendedAnnotationCnt* will have an initial 3-character format to preserve the order among the integer values even after converting them into strings[1]. In Figure 4(d) Step 1, we illustrate the itemization of the *ClassBird1* summary object attached to tuple $r1$.

○ **Indexing:** The text values generated from the Itemization step will be inserted into the Summary-BTree index. The index follows the same structure and operations of the standard B-Tree. And hence, the B-Tree's maintenance algorithms, i.e., insertion and deletion, are all leveraged in the Summary-BTree index. The indexed values will appear in the leaf nodes of the index sorted alphabetically as depicted in the figure. The only exception compared to the standard B-Tree will be in the heap pointers stored in the leaf nodes, which are called *backward pointers* and described next.

○ **Backward Referencing:** We make use of the fact that the storage of the annotation summaries is entirely transparent from (and not directly query-able by) the end-users. Hence, we have the opportunity to optimize the internal structure of the proposed tables and indexes for efficient performance. A key trick in the Summary-BTree index is that the leaf nodes will point back to their annotated data tuples in Relation R instead of pointing back to the *R_SummaryStorage* table. For example, the index entries `"Disease:002"` and `"Disease:008"` will point back to tuple $R.r2$ and $R.r1$, respectively. These backward pointers will be created and maintained under the different operations as described in sequel [2].

The advantages of the Summary-BTree index are two fold: (1) It builds on the existing storage scheme of InsightNotes without the need to replicate or normalize the summary objects, and hence the optimized propagation performance is not affected. And (2) As the experimental evaluation will confirm (Section 6), the backward-referencing mechanism achieves up to 11x speedup in query performance compared to the baseline indexing scheme.

4.1.2 Summary-BTree Index Operations

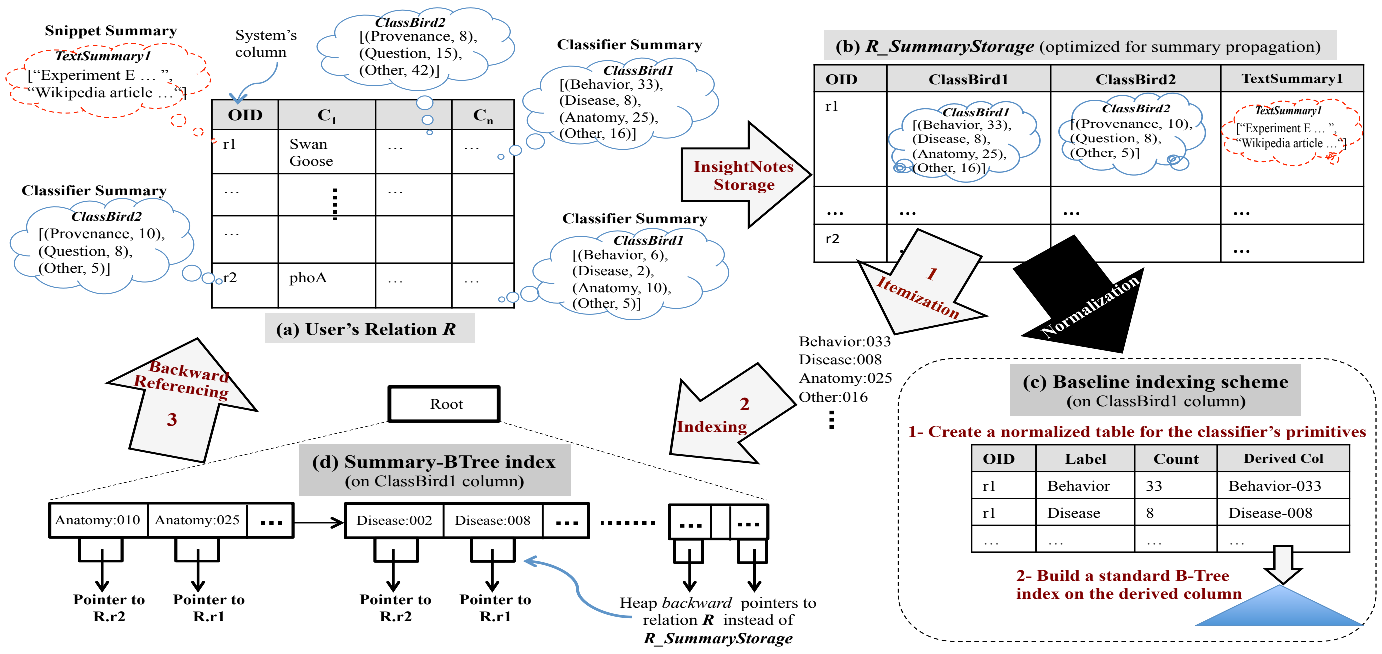
To enable the backward-referencing mechanism, we developed an internal function, called *diskTupleLoc()*, within the database engine, which takes a tuple's identifier (OID) and returns its heap location. This function will be used inside the index's maintenance algorithms to create the correct backward pointers. Notice that this mechanism does not break the transparency concept in database systems since it is entirely encapsulated within the index structure and not exposed to the outside world (neither end-users nor database developers). The index is maintained under the following operations:

○ **Adding Annotation−Insertion:** Adding a new annotation on an un-annotated tuple in R results in inserting a new tuple in *R_SummaryStorage*. The system will then retrieve the heap location of the data tuple, itemize the indexed classifier summary object, and insert them into the Summary-BTree index as illustrated in Figure 4(d).

○ **Adding Annotation−Update:** Adding a new annotation on an already-annotated tuple in R will result in updating the corresponding summary objects in *R_SummaryStorage*. For example, if a new annotation highlighting a disease is added to $R.r1$, then the *ClassBird1*'s summary object will be updated by incrementing the count

[1]If the number of annotations assigned to a single classifier's label exceeds 999, then InsightNotes automatically increments the number of allocated characters and re-builds the index. However, it is a very rare operation.

[2]The SummaryStorage tables are still directly accessible using SQL queries, but for administrative tasks only. Such administrative queries use table-scan plans instead of index-scan plans.



Snippet Summary
TextSummary1
["Experiment E … ", "Wikipedia article …"]
System's column
ClassBird2
[(Provenance, 8), (Question, 15), (Other, 42)]
Classifier Summary
ClassBird1
[(Behavior, 33), (Disease, 8), (Anatomy, 25), (Other, 16)]
OID
C1
Cn
r1
Swan Goose
r2
phoA
Classifier Summary
ClassBird2
[(Provenance, 10), (Question, 8), (Other, 5)]
Classifier Summary
ClassBird1
[(Behavior, 6), (Disease, 2), (Anatomy, 10), (Other, 5)]
(a) User's Relation R
InsightNotes Storage
(b) R_SummaryStorage (optimized for summary propagation)
OID
ClassBird1
ClassBird2
TextSummary1
1 Itemization
Normalization
Behavior:033
Disease:008
Anatomy:025
Other:016
2 Indexing
Backward Referencing 3
Root
(d) Summary-BTree index (on ClassBird1 column)
Anatomy:010
Anatomy:025
Disease:002
Disease:008
Pointer to R.r2
Pointer to R.r1
Heap backward pointers to relation R instead of R_SummaryStorage
(c) Baseline indexing scheme (on ClassBird1 column)
1- Create a normalized table for the classifier's primitives
2- Build a standard B-Tree index on the derived column

OID	Label	Count	Derived Col
r1	Behavior	33	Behavior-033
r1	Disease	8	Disease-008
…	…	…	…

Figure 4: Summary-Btree Index For Indexing The *Classifier* Type Summary Objects.

of the *Disease* class label to be 9. To update the index, the system will trigger a deletion and then re-insertion only for the modified class label—The other class labels within the object will remain untouched. For example, a deletion of key `"Disease:008"` and insertion of key `"Disease:009"` will take place.

◦ **Deleting Annotation or Tuple:** The deletion of an annotation will result in updating the corresponding summary objects in *R_SummaryStorage*. Therefore, the same procedure described above will be applied. Similarly, the deletion of a data tuple from R will result in deleting the corresponding tuple in *R_SummaryStorage*, and all its index entries will be deleted.

◦ **Summary-BTree Querying:** To answer an equality query over the index, e.g., `"classLabel = constant"`, a probing key will be formed by concatenating the two operands, i.e., `"classLabel:Extended_constant"`, where `Extended_constant` is the 3-character format of the constant value. In the case of a range query, e.g., `"constant1 > classLabel > constant2"`, two probing keys will be formed; a *starting key* as `"classLabel:Extended_constant1"`, and a *stopping key* as `"classLabel:Extended_constant2"`. All the keys in between will lead to the qualifying data tuples. If either of the starting or stopping keys is missing, then it will be replaced by `"classLabel:000"`, or `"classLabel:999"`, respectively.

4.1.3 *Summary-BTree Theoretical Bounds*

The Summary-BTree inherits the efficient logarithmic performance from the B-Tree index since they have similar structure. The following Theorem states the theoretical bounds of the index.

Theorem: *Assuming that the number of data tuples in the user's relation R is M, the number of Classifier-type summary objects is N, the number of class labels per summary object is k, and the disk page size in records is B, then the following theoretical bounds hold for a Summary-BTree index:*

◦ *Adding Annotation—Insertion is O($kLog_BkN + Log_BM$)*

◦ *Adding Annotation—Update is O($2Log_BkN + Log_BM$)*

◦ *Deleting data tuple is O($kLog_BkN + Log_BM$)*

◦ *Equality search is O(Log_BkN)* □

Proof: Assuming N summary objects and each object has k class labels, then the number of indexed keys is O(kN). Therefore, any single search, insertion or deletion will be bounded by O(Log_BkN). When adding a new annotation that triggers a new insertion in the SummaryStorage table, the k class labels will be inserted into the index which will cost O($kLog_BkN$). In contrast, if the added annotation will trigger an update of an existing class label, then only that label is deleted and then re-inserted, which will cost O($2Log_BkN$). Finally, when inserting into or deleting from the index tree, the system needs to retrieve the heap location of the data tuple. This operation uses a B-Tree index on the OID column in R with the cost of O(Log_BM).

5. SUMMARY-BASED QUERY OPTIMIZATION

When a query involves both the summary-based and the standard SQL operators, then the traditional transformation and equivalence rules alone will be of a very limited use. This is because the semantics of the new operators are unknown to current optimizers. For example, the current optimizer may not be able to use the standard *selection-pushdown* rule to push a selection operator below a join operator because there is a summary-based operator in-between. Another example is illustrated in Figure 5(a), where the query plan involves a summary-based sort $\mathcal{O}$ and selection $\mathcal{S}$ operators on top of a traditional join operator $\bowtie$. In this case the current optimizers cannot apply any of the known transformation rules to create equivalent query plans. In this section, we introduce several important equivalence rules involving the summary-based operators, and extend the query optimizer to leverage them and create a larger pool of possible query plans.

5.1 Extended Equivalence Rules

• **Rules for Summary-Based Selection ($\mathcal{S}_p(R)$):** Few important rules involving the $\mathcal{S}$ operator include:

$$\mathcal{S}_p(\sigma_c(R)) = \sigma_c(\mathcal{S}_p(R)) \quad (1)$$

$$\mathcal{S}_p(R \bowtie_c S) = \mathcal{S}_p(R) \bowtie_c S, \text{ iff } p \text{ is on instances in } R \text{ not in } S. \quad (2)$$

Proof: Rule 1 is correct since neither the σ operator changes the summaries' content nor the $\mathcal{S}$ operator changes the data's content. And thus, the commutativity property between the two operators apply. This rule enables the system to switch the order of predicates and use the available indexes—either on the data or the summaries—as needed. Rule 2 enables pushing the summary-based selection operator before the join operator. Rule 2 is correct since predicates p are on instances linked only to one of the two relations, say R. Therefore, when the $\bowtie$ operator merges the summary objects attached to the joined tuples, the summary objects related to p are guaranteed not to change since they have no counterparts on S. And hence, the rule applies.

• **Rules for Summary-Based Sort ($\mathcal{O}_{f[,direction]}(R)$):** We focus on an important case where an existing Summary-BTree index can provide R's tuples in an *interesting order* to the query, and hence the sort operator can be eliminated. The following rules state that the order of R's tuples is preserved under certain transformations. We use notation $\overline{R}^L$ to indicate that R has an *interesting order* w.r.t a classifier instance L.

$$\sigma_c(\overline{R}^L) = \overline{\sigma_c(R)}^L \quad (3)$$

$$\mathcal{S}_p(\overline{R}^L) = \overline{\mathcal{S}_p(R)}^L \quad (4)$$

$$\overline{R}^L \bowtie S = \overline{R \bowtie S}^L, \text{ iff } \bowtie \text{ preserves } R\text{'s order, and L is not on } S. \quad (5)$$

$$\mathcal{J}(\overline{R}^L, S) = \overline{\mathcal{J}(R,S)}^L, \text{ iff } \mathcal{J} \text{ preserves } R\text{'s order, and L is not defined on } S. \quad (6)$$

Proof: Rules 3 and 4 indicate that the selection operators (σ and $\mathcal{S}$) do not change the interesting order of R and preserve it in the output. This is guaranteed since these operators do not change the content of their summaries. For the join operators (Rules 5 and 6), the order w.r.t L is preserved only if two conditions are met: (1) The join algorithm preserves R's order, e.g., R is the outer relation of the join, and (2) Relation S does not have the summary instance L defined on it. If the 2^{nd} condition is not met, then the join operators ($\bowtie$ or $\mathcal{J}$) would merge the summary objects of L, and thus the order may not be preserved. Otherwise, Rules 5 and 6 also applies.

Example 4: *Assume a query Q that joins Relation R depicted in Figure 4 with another relation $S(c1, c2)$ based on data attributes $R.c_1 = S.c_1$. Then, Q selects only the tuples with more than five disease annotations, i.e.,* `ClassBird1.disease > 5`*, and produces the output sorted by the count of these disease annotations. An initial query plan based on the sequence presented above is illustrated in the following figure (Figure 5(a)). Then, consider the following two cases:*
<u>*Case I*</u>*: Relation S has the* ClassBird1 *summary instance defined on it. In this case, the summary-based selection operator cannot be pushed below the join operator, and the system will use the initial plan in Figure 5(a).*
<u>*Case II*</u>*: Relation S does not have the* ClassBird1 *summary instance defined on it. In this case, the system will use Rule 2 to push the summary-based selection operator before the join. And assuming that* ClassBird1 *summary instance on R is indexed, then the index can be used to retrieve the tuples with more than five disease annotations (in a sorted order). Then, based on Rule 5, the join operator preserves the order of the tuples, and hence the summary-based sort operator can be removed as illustrated in Figure 5(b).*

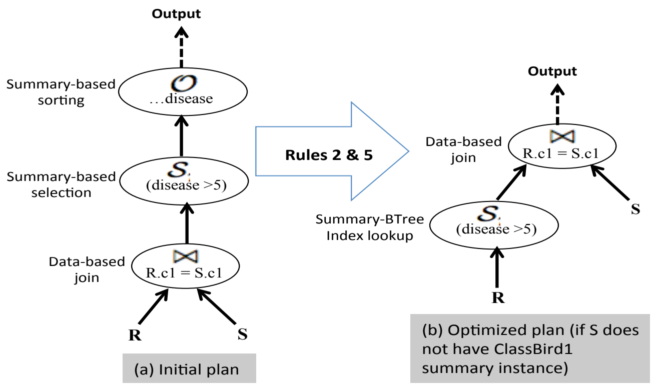


Figure 5: Rule-Based Equivalent Plans in InsightNotes.

• **Rules for Summary-Based Filter ($\mathcal{F}_p(R)$):** Few important rules involving the $\mathcal{F}$ operator include:

$$\mathcal{F}_p(R \bowtie_c S) = \mathcal{F}_p(R) \bowtie_c S, \text{ iff } p \text{ is on instances in } R \text{ not in } S. \quad (7)$$

$$\mathcal{F}_p(R \bowtie_c S) = \mathcal{F}_p(R) \bowtie_c \mathcal{F}_p(S), \text{ iff } p \text{ is structural predicate.} \quad (8)$$

Proof: Rules 7 and 8 address pushing the filter operator before the join. Both rules aim for eliminating unnecessary summary objects—and hence their processing cost in the query pipeline—as early as possible. Rule 7 can be proved in similar way to Rule 2, i.e., the $\bowtie$ operator is guaranteed not to alter the summary objects related to predicate p since they are attached to only relation R. Similarly, the $\mathcal{F}$ operator does not change the data's content, and hence the join predicates c are not affected. Therefore Rule 7 applies.

Rule 8 indicates that if the predicate is *structural*—A *structural* predicate is defined as a predicate on the *InstanceID* or the *SummaryType* of the summary object—then p can be pushed to both sides before the join operation. For example, referring to Figure 4, if a query is interested only in the summary objects of instance *ClassBird1*, then all other summaries of instances *ClassBird2* and *TextSummary1* can be dropped as early as possible. Rule 8 can be proved in the same way as Rule 7.

• **Rules for Summary-Based Join ($\mathcal{J}_p(R, S)$):** Few important rules involving the $\mathcal{J}$ operator include:

$$\sigma_c(\mathcal{J}_p(R,S)) = \mathcal{J}_p(\sigma_c(R), S), \text{ iff } c \text{ is on } R\text{'s attributes.} \quad (9)$$

$$\mathcal{S}_{p1}(\mathcal{J}_{p2}(R,S)) = \mathcal{J}_{p2}(\mathcal{S}_{p1}(R), S), \text{ iff } p1 \text{ is on instances in } R \text{ not in } S. \quad (10)$$

$$T \bowtie_c \mathcal{J}_p(R,S) = \mathcal{J}_p((T \bowtie_c R), S), \text{ iff } p \text{ is on instances not in } T \text{ and } c \text{ does not involve } S\text{'s attributes.} \quad (11)$$

Proof: Rules 9 and 10 address pushing the selection operators (σ or $\mathcal{S}$) before the summary-based join operator whenever possible. It is always a valid transformation in the case of the σ operator as long as the predicates c are on one of the two relations (Rule 9). This is because the σ and the $\mathcal{J}$ operate on disjoint pieces of the tuple, i.e., the data values, and the summaries, respectively. Rule 10 is correct since the summary objects related to $p1$ are only attached to relation R. And thus, the $\mathcal{J}$ is guaranteed not to alter these objects after the join. Rule 11 states the conditions for switching the order between summary- and data-based join operators. The order can be switched iff the summary-based join predicates p involve instances not defined on T. And thus, joining early with T ($T \bowtie_c R$) is guaranteed not to affect the evaluation of p.

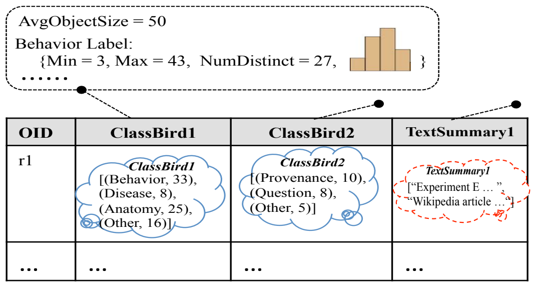

AvgObjectSize = 50
Behavior Label:
{Min = 3, Max = 43, NumDistinct = 27, }
......
OID
ClassBird1
ClassBird2
TextSummary1
r1
ClassBird1
[(Behavior, 33), (Disease, 8), (Anatomy, 25), (Other, 16)]
ClassBird2
[(Provenance, 10), (Question, 8), (Other, 5)]
TextSummary1
["Experiment E … ", "Wikipedia article …"]
...

Figure 6: Example of Classifier-Type Maintained Statistics.

5.2 Cost Model and Cardinality Estimation

Statistics Collection: The equivalence rules presented in Section 5.1 enable the query optimizer to generate a larger pool of equivalent query plans. The next step is to estimate the cost of the new summary-based operators in order to select the cheapest plan. Towards this goal, InsightNotes maintains several statistics over the summary objects attached to a given relation R. These statistics are similar to those maintained by traditional DBMSs except that they capture the internal semantics of the summary objects.

Demonstrating over an example, assume relation R has three summary instances linked to it as illustrated in Figure 6. Then, for each summary instance (one column in Figure 6), InsightNotes maintains the average object size (`AvgObjectSize`). In the case of Classifier-type objects, e.g., *ClassBird1* and *ClassBird2*, the size is fixed for all objects within one instance. In contrast, for the Snippet-type and Cluster-type objects, the size may differ from one object to another. Moreover, the system maintains several statistics for each classifier label within the Classifier-type objects. For example, for *ClassBird1*, four data structures are maintained—one for each class label. Each data structure holds some statistics on the *count* field associated with that label, which include `{Min, Max, NumDistinct, Equi-Width Histogram}` as depicted in Figure 6. These statistics are maintained whenever a summary object is updated.

Cardinality and Cost Estimation: To avoid re-inventing the wheel, the new summary-based operators leverage the same heuristics that the standard SQL operators use to estimate their cardinalities and costs. For example, the filter operator $\mathcal{F}$ uses the same heuristics as the standard projection operator π, e.g., based on the *AvgObjectSize* statistics, the $\mathcal{F}$ operator estimates the size of the new tuples and the number of needed disk blocks. Similarly, the summary-based selection operator $\mathcal{S}$ uses the same heuristics as the standard selection operator σ, e.g., referring to the $\mathcal{S}$ operator in Example 4, the system uses the maintained statistics (`{Min, Max, NumDistinct, Equi-Width Histogram}`) over the *ClassBird1.Disease* label to estimate the number of output tuples having more than 5 disease-related annotations. Moreover, if a Summary-BTree index is used to answer this predicate, then the number of performed I/Os can be estimated based on the index's theoretical bounds.

The summary-based join operator $\mathcal{J}$ also follows the same heuristics as the standard join operator $\bowtie$, e.g., the size of joining two relations R and S based on an equality join on `ClassBird2.Provenance` can be estimated by multiplying the size of both relations, and then dividing by the largest value between the `NumDistinct` statistics on `ClassBird2.Provenance` from both sides. Currently, InsightNotes supports only two implementation choices for the $\mathcal{J}$ operator, which are either a block nested-loop join, or an index-based join.

6. EXPERIMENTS

The proposed extensions are implemented within the InsightNotes prototype engine [22], which is based on the open-source PostgreSQL DBMS. The experiments are conducted using an AMD Opteron Quadputer compute server with two 16-core AMD CPUs, 128GB memory, and 2 TBs SATA hard drive.

Application Datasets: We use annotated database that stores information related to 10s of thousands of birds worldwide. The largest annotated table in the database is the *Birds* table that stores the birds' basic information. The table consists of 45,000 tuples, each consisting of 12 attributes, e.g., scientific name, Ids across different systems, description, genus, family, and habit. The table size in the database is approximately 450MBs. The collected number of annotations is approximately 9x10^6 annotations describing a wide range of bird related information, e.g., color, body shape or weight, certain behavior or sound, eating habits, geographic location, or observed diseases. The size of each annotation varies between 150 and 8,000 characters. The total size of the raw annotations table (the 9x10^6 annotations) is around 5GBs.

Summarization Techniques: InsightNotes has several integrated data mining techniques for annotation summarization, e.g., the Naive Bayes [10] technique for annotation classification, the CluStream technique [2] for incremental clustering of annotations, and the LSA (Latent Semantic Analysis) technique [18] for text summarization and snippet creation. For the purpose of our experiments, we link the *Birds* table with two summary instances: (1) A Classifier summary instance *ClassBird1* that classifies each annotation to one of the labels: {*'Disease', 'Anatomy', 'Behavior', 'Other'*}, and (2) A Snippet summary instance *TextSummary1* that summarizes each annotation larger than 1,000 characters and creates a snippet that has a maximum of 400 characters. We then create a Summary-BTree index over *ClassBird1*.

Index Creation Overhead: The first set of experiments study the overheads associated with the creation of the summary-based index (Figures 7, 8, and 9). In the experiments, we vary the number of annotations (over the x-axis) between 450x10^3 (corresponding to 10 annotations per tuple on average), to 9x10^6 (corresponding to 200 annotations per tuple on average). Figure 7 illustrates the storage overhead of both the *Baseline* and *Summary-BTree* schemes discussed in Section 4.1. In the former scheme, the summary objects are replicated and stored in a normalized form, and then a standard B-Tree index is created over them. In contrast, in the latter scheme, a Summary-BTree index is created over the de-normalized representation of the summary objects. As the results show, the index size in both cases is almost the same. However, the proposed *Summary-BTree* scheme saves up to 65% of the storage overhead as it requires no replication of the data. The results also show that the storage overhead is almost fixed under the different sizes of the raw annotations. The reason is that once each data tuple has an attached classifier summary object, then the number and size of the summary objects becomes fixed and will not change. The increase in the number of annotations only changes the integer value assigned to the class labels, which does not affect the size.

In Figures 8 and 9, we measure the time overhead of creating the indexes in *bulk* and *incremental* modes, respectively. In the *bulk* mode (Figure 8), the raw annotations and the summary objects will be first created, and then the indexes will be built. This is the recommended mode for initial uploading of large datasets into the database. We measured, over the y-axis, the relative time of creating the index to the time of uploading the raw annotations and creating the summary objects. The indexing time under the *Summary-BTree* scheme involves the time for itemization, insertion

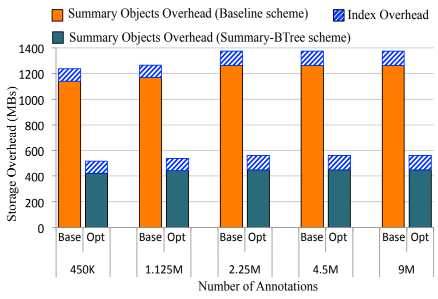


Figure 7: Storage Overhead.

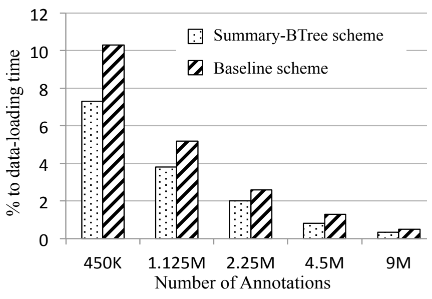


Figure 8: Bulk Index Creation.

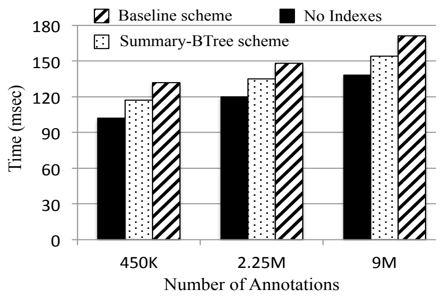


Figure 9: Incremental Indexing.

of indexed keys, and computing the backward references. For the *Baseline* scheme, the indexing time includes the de-normalization and storage in other tables, and the insertion of the indexed keys. The figure illustrates that the creation of the Summary-BTree index is more efficient than the baseline index by up to 35%.

The performance of the incremental indexing is studied in Figure 9. We considered the cases of inserting annotations with: (1) No indexes, (2) A Summary-BTree index, and (3) A Baseline B-Tree index. For each data point in the figure, we insert 100 annotations, measure the insertion time of each annotation under the three cases, and then report the average over the 100 insertions. As the figure shows, the indexing overhead using the Summary-BTree index is approximately 10% to 15% of the insertion time, while the baseline indexing scheme has around 20% to 37% overhead due to the de-normalization step.

Query Performance: The next set of experiments study the effect of utilizing the Summary-BTree index to speedup queries involving summary-based predicates (Figures 10, 11, 12, and 13). The results in Figure 10 illustrate the performance gain from the Summary-BTree index using a Select-Project (SP) query, where the selection predicate is in the form of: `"r.$.getSummaryObject('ClassBird1').getLabelValue('Disease') = constant"`. The query's response time is presented in the y-axis (in Log scale) under three cases: (1) using no indexes, (2) using the *Baseline* standard B-Tree index, and (3) using the Summary-BTree index. We experimented with different query selectivities, i.e., 0.1%, 1%, and 5%, and the differences were minor in each case. Therefore, in Figure 10 we report the results of only the 1% selectivity (around 450 data tuples). The figure illustrates that the Summary-BTree index has approximately 3x speedup over the baseline index. This is because the latter index involves more levels of indirection, and hence requires more join operations to reach the desired data tuples. As expected both indexes achieve around two orders of magnitude speedup compared to the *NoIndex* case.

The experiment in Figure 11 studies the performance of a Select-Project (SP) query involving two conjunctive predicates: (1) A range predicate selecting the tuples having a number of anatomy-related annotations within a given range, i.e., `"r.$.getSummaryObject('ClassBird1').getLabelValue('Anatomy') in [x,y]"`, and (2) A keyword search predicate over the text summarization instance, i.e., `"r.$.getSummaryObject('TextSummary1').containsUnion(kw1, ...)"`. When the index scan over *ClassBird1* is disabled (the *NoIndex* case), InsightNotes uses a table scan followed by a summary-based selection operator $\mathcal{S}$ to apply both predicates. In contrast, when the index scan is enabled, InsightNotes uses the index to evaluate the range predicate and on top of that a $\mathcal{S}$ operator to apply the keyword search predicate. The results illustrate that the *Summary-BTree* index is around 2x faster than the baseline index.

It is worth noting that in the previous experiments, the *Baseline* indexing scheme is used only to evaluate the selection predicates involved in the query. Yet, the for the summary propagation purpose to end-users, InsightNotes still reads the summary objects from its de-normalized storage, i.e., *R_SummaryStorage* (Refer to Figure 4). And hence, both indexes do not pay the cost of building the summary objects from their primitive components. To confirm that depending only on the *Baseline* scheme (the normalized storage of summary objects) can significantly slowdown the summary propagation, we performed the experiment in Figure 12. In the experiment, we used the same query as in Figure 11 and compared between the two indexing schemes. The only difference is that the *Baseline* scheme in this experiment will not only evaluate the predicates, but also form the summary objects for propagation. In this case, the *Baseline* scheme showed around 7x slower performance compared to the Summary-BTree indexing scheme.

In Figure 13, we study the effectiveness of augmenting the Summary-BTree index with backward pointers that point directly to the annotated data tuples instead of the conventional pointers that point to the indexed objects. We use the same SP query used in Figures 10. In the experiments, we consider four cases. The first case is that the index uses the backward pointers, and the annotation summaries are propagated along with the query's result (labeled `Backward-Propagation`). The second case is that the index uses the backward pointers, and the annotation summaries are not propagated along with the query's result (labeled `Backward-NoPropagation`). The other two cases are the same of the above except that the index uses the conventional pointers instead of the backward pointers, i.e., the Summary-BTree index pointers will point to the *ClassBird1* summary objects. The results in Figure 13 show that propagating the annotation summaries has almost the same cost under both the backward and conventional pointers. The reason is that the join operation between the data table and its SummaryStorage table has a 1-1 cardinality, and hence the performance is very similar regardless of which table is used as the outer table in the join. In contrast, if the summary propagation is not required, then the backward pointers will save unnecessary join with the SummaryStorage table, which achieves up to 4x speedup in query execution.

Effectiveness of Query Optimization and Transformation Rules: In Figures 14 and 15, we study the effect of some of the new transformation rules and query optimizations proposed in Section 5. The first experiment (Figure 14) measures the performance of the query demonstrated in Example 4 in Section 5. Relations R and S in the rules correspond to the *Birds* and *Synonyms* tables, respectively. The *Synonyms* table consists of approximately 225,000 tuples and linked to the *Birds* table in a many-to-one relationship.

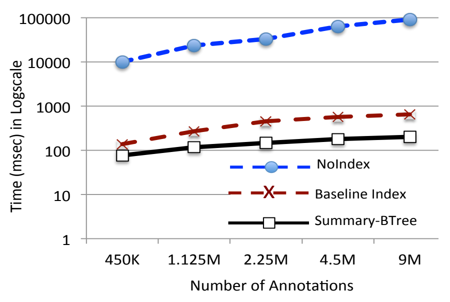


Figure 10: Index vs. No Index (SP Query)

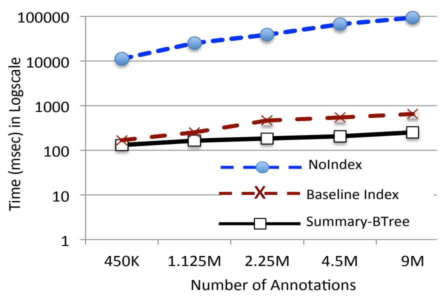


Figure 11: Two-Predicates SP Query

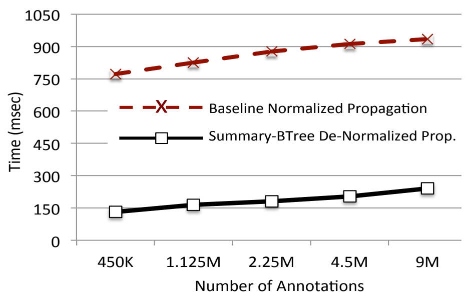


Figure 12: De-Normalized Propagation.

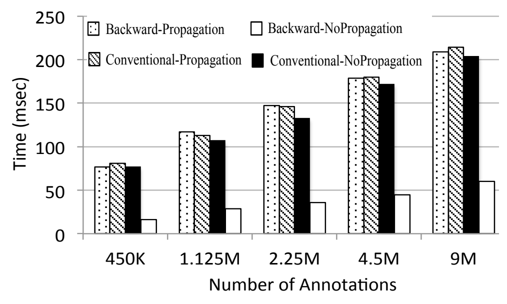


Figure 13: Effectiveness of Backward Ptrs.

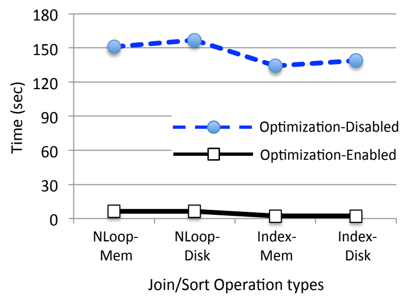


Figure 14: Optimization Rules {2, 5}.

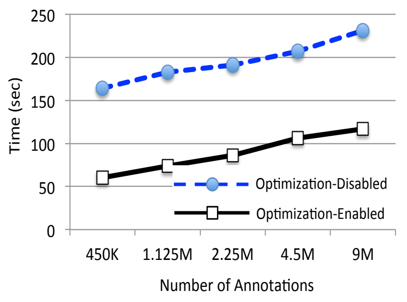


Figure 15: Optimization Rule {11}.

Only the *TextSummary1* instance is linked to the *Synonyms* table, and hence Optimization Rules 2 and 5 can be applied. The experiment compares the response time of the default query plan (Figure 5(a)) against that of the optimized query plan (Figure 5(b)). We set the dataset size to 9x10^6 annotations, and we consider two cases for each of the join and sort operators as illustrated in the x-axis. The join operator either uses an index-based algorithm with an index on the join column in S (labeled `Index`), or a block nested-loop join algorithm (labeled `NLoop`), and the sort operator either uses a memory-based (labeled `Mem`) or disk-based (labeled `Disk`) sort algorithms. The figure illustrates the effectiveness of the transformation and optimization rules in all of the four cases to speedup the query's response time by a factor of 15x.

In Figure 15, we study the effectiveness of Optimization Rule 11, where the order between data- and summary-based join operators can be switched. Relations R and S correspond to the same tables as in the previous experiment, and the summary-based join between them involves a summary-based keyword search on their combined *TextSummary1* summary objects—No summary-based index can be used in this case. Relation T is a replica to relation R, and hence they have a 1-1 relationship through an indexed column for the birds' unique identifiers. With no optimizations, the default plan performs the $\mathcal{J}(R, S)$ operation first using a block nested-loop join, and then performs the data-based join ($\bowtie$) with T. In contrast, the optimized plan switches the join order to make use of the available index on the birds' identifiers in T. Thus, the join operation $R \bowtie T$ is performed first using an index-based join, and then the results is summary-based joined ($\mathcal{J}$) with S. The performance results in Figure 15 indicate that the optimized plan achieves around 3.5x speedup compared to the default plan.

Usability Case Study: Similar to the motivating example presented in Section 1.1, we performed a usability case study to show direct impact of the newly added features on users' experience. We formed a team of 20 students divided into two groups, where one group uses the basic InsightNotes engine while the other group uses the extended system (called *InsightNotes+*). Each student will answer each of the three queries highlighted in Figure 16. In the figure, we report the average time taken by each group (including the time of writing the query), and the results' accuracy.

Query Semantics	# Qualifying data tuples	InsightNotes Group	InsightNotes+ Group
Q1: Report the data tuples sorted based on the number of attached disease-related annotations	100	Time: 5.2 min Accuracy: 100%	Time: 40 sec Accuracy: 100%
Q2: Join version 1 of the data (V1) with version (V2) and report the same objects, i.e., V1.ID = V2.ID, having different number of provenance-related annotations	5	Time: 8.1 min Accuracy: 100% ***(Reports 450 Tuples)***	Time: 54 sec Accuracy: 100%
Q3: Select the birds' records having more than 3 question-related annotations	10	--- ***(Reports 45K Tuples)***	Time: 52 sec Accuracy: 100%

Figure 16: Usability Case Study.

As the results show, both groups are able to answer $Q1$ and $Q2$ queries with 100% accuracy. However, the *InsightNotes* group took significantly longer time to produce the results—which may not be acceptable in many applications. The reason is that InsightNotes cannot fully answer any of these queries, and thus a manual effort is needed to post-process the answer produced from InsightNotes. For example, in $Q1$ the students need to manually sort the 100 data tuples based on the number of their disease-related annotations (summary-based sorting), while in $Q2$, they needed to go over the joined tuples (based on the ID data columns)—which are 450 tuples—and manually check the second join predicate (based on the number of provenance-related annotations) and report the 5 qualifying tuples. For $Q3$, since InsightNotes cannot apply a summary-based selection operation, all the data tuples (45,000) will be reported, and it is impractical to manually select the desired tuples from them. On the other hand, the *InsightNotes+* group is able to answer the three queries in few seconds.

7. RELATED WORK

Annotation management has been extensively studied in the context of relational DBMSs [4, 9, 14, 15, 21]. Several of these systems focus on extending the relational algebra and query semantics for propagating the annotations along with the queries' answers [4, 9, 14, 21]. The Mondrian system [14] has proposed extensions to treat the annotations as first-class citizens, where users can query and manipulate the annotations through newly defined operators. Other systems address the annotation propagation in the context of containment queries [21], logical views [7], or automated copying to newly inserted data [11, 17]. The systems proposed in [8, 13] support special types of annotations, e.g., treating annotations as data and annotating them [8], and capturing users' beliefs as annotations [13]. All of these systems share a common limitation, which is that they all manipulate the raw annotations. Therefore, they do not provide any support for summarizing, extracting useful knowledge, or applying analytics over the raw annotations. The InsightNotes system and its extensions proposed in this paper address such limitations, and enable end-users to query the annotation summaries in novel ways, which otherwise were not possible.

In the domains of e-commerce, social networks, and entertainment systems, e.g., [12, 19], the annotations are usually referred to as *tags*. These systems deploy advanced mining and summarization techniques for extracting the best insight possible from the annotations to enhance users' experience. They use such extracted knowledge to take actions, e.g., providing recommendations and targeted advertisements. However, unlike relational DBs, the retrieval mechanisms in these systems are typically straightforward and do not involve complex processing or transformations, i.e., no advanced query processing is required over the annotations summaries once created. Therefore, these techniques do not address the complex query processing and optimization challenges prevalent to scientific relational DBs that are addressed in this paper.

Scientific systems and workflows have also leveraged the concept of semantic and ontology-based annotations, e.g., [3, 6]. These systems use semantic annotations to either summarize complex workflows [3], or help in building and verifying workflows [6]. These systems are based on process-centric annotations, e.g., annotations capturing the semantics of each function in a workflow, the structure of their input and output arguments, etc. In contrast, InsightNotes manages data-centric annotations that are independent from how the data is processed. Nevertheless, the proposed summary-based query operators, access methods, and optimizations are all new and have not beed addressed in current systems.

8. CONCLUSION

The large volume, increasing complexity, and hidden semantics of the emerging annotation repositories in modern applications create unprecedented challenges to annotation management techniques. In this paper, we proposed extensions to the *InsightNotes* system for elevating the annotation summaries from being *"propagate-only"* objects to be *"first-class"* citizens. Hence, it becomes feasible for applications to express complex queries over both the data and their attached annotation summaries, which otherwise is not possible. The key contributions include: (1) Proposing manipulation functions and query operators to seamlessly operate on the summary objects at query time, (2) Developing specialized summary-based indexing scheme and access methods for efficient predicate evaluation and retrieval of the summary objects, and (3) Introducing an extended query optimizer that enables advanced optimizations for queries involving both the summary-based and the standard query operators. The extensions are implemented within the InsightNotes prototype engine, and the results have demonstrated the practicality and efficiency of the proposed extensions and techniques w.r.t both the system's performance, and users' experience.

As part of future work, we plan to enrich the system with more implementation choices for the summary-based operators, enable multi-level (hierarchical) summarization, and extend the querying mechanisms over the multi-level model.

9. REFERENCES

[1] eBird Trail Tracker Puts Millions of Eyes on the Sky. `https://www.fws.gov/refuges/RefugeUpdate/MayJune_2011/ebirdtrailtracker.html`.

[2] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu. A Framework for Clustering Evolving Data Streams. In *VLDB*, pages 81–92, 2003.

[3] P. Alper, K. Belhajjame, C. Goble, and P. Karagoz. Small Is Beautiful: Summarizing Scientific Workflows Using Semantic Annotations. In *IEEE BigData Congress*, pages 318–325, 2013.

[4] D. Bhagwat, L. Chiticariu, and W. Tan. An annotation management system for relational databases. In *VLDB*, pages 900–911, 2004.

[5] C. Bogdanschi and S. Santini. An annotation database for multimodal scientific data. In *Proc. SPIE 7255, Multimedia Content Access: Algorithms and Systems III*, pages 307–314, 2009.

[6] S. Bowers and B. LudŁscher. A Calculus for Propagating Semantic Annotations through Scientific Workflow Queries. In *In Query Languages and Query Processing (QLQP)*, 2006.

[7] P. Buneman and et. al. On propagation of deletions and annotations through views. In *PODS*, pages 150–158, 2002.

[8] P. Buneman, E. V. Kostylev, and S. Vansummeren. Annotations are relative. In *Proceedings of the 16th International Conference on Database Theory*, ICDT '13, pages 177–188, 2013.

[9] L. Chiticariu, W.-C. Tan, and G. Vijayvargiya. DBNotes: a post-it system for relational databases based on provenance. In *SIGMOD*, pages 942–944, 2005.

[10] P. R. Christopher D. Manning and H. Schutze. Book Chapter: Text classification and Naive Bayes, in Introduction to Information Retrieval. In *Cambridge University Press*, pages 253–287, 2008.

[11] M. Eltabakh, W. Aref, A. Elmagarmid, and M. Ouzzani. Supporting annotations on relations. In *EDBT*, pages 379–390, 2009.

[12] A. Gattani and et. al. Entity extraction, linking, classification, and tagging for social media: a wikipedia-based approach. *Proc. VLDB Endow.*, 6(11):1126–1137, 2013.

[13] W. Gatterbauer, M. Balazinska, N. Khoussainova, and D. Suciu. Believe it or not: adding belief annotations to databases. *Proc. VLDB Endow.*, 2(1):1–12, 2009.

[14] F. Geerts and et. al. Mondrian: Annotating and querying databases through colors and blocks. In *ICDE*, pages 82–93, 2006.

[15] F. Geerts and J. Van Den Bussche. Relational completeness of query languages for annotated databases. In *Proceedings of the 11th international conference on Database Programming Languages (DBPL)*, pages 127–137, 2007.

[16] K. Ibrahim, D. Xiao, and M. Eltabakh. InsightNotes+: Advanced Query Processing in Summary-Based Annotation Management. Technical Report WPI-TR-14-05: http://web.cs.wpi.edu/∼meltabakh/TR-14-05.pdf.

[17] Q. Li, A. Labrinidis, and P. K. Chrysanthis. ViP: A User-Centric View-Based Annotation Framework for Scientific Data. In *Proceedings of the 20th international conference on Scientific and Statistical Database Management (SSDBM)*, pages 295–312, 2008.

[18] A. Nenkova and K. McKeown. A Survey of Text Summarization Techniques. In *Book: Mining Text Data*, pages 43–76, 2012.

[19] A. Rae, B. Sigurbjörnsson, and R. van Zwol. Improving tag recommendation using social networks. In *Adaptivity, Personalization and Fusion of Heterogeneous Information*, RIAO, pages 92–99, 2010.

[20] M. Stonebraker and et. al. Data Curation at Scale: The Data Tamer System. In *CIDR*, 2013.

[21] W.-C. Tan. Containment of relational queries with annotation propagation. In *DBPL*, 2003.

[22] D. Xiao and M. Y. Eltabakh. InsightNotes: Summary-Based Annotation Management in Relational Databases. In *SIGMOD Conference*, pages 661–672, 2014.

Estimating Data Integration and Cleaning Effort

Sebastian Kruse
Hasso Plattner Institute (HPI), Germany
sebastian.kruse@hpi.de

Paolo Papotti
Qatar Computing Research Institute (QCRI), Qatar
ppapotti@qf.org.qa

Felix Naumann
Hasso Plattner Institute (HPI), Germany
felix.naumann@hpi.de

ABSTRACT

Data cleaning and data integration have been the topic of intensive research for at least the past thirty years, resulting in a multitude of specialized methods and integrated tool suites. All of them require at least some and in most cases significant human input in their configuration, during processing, and for evaluation. For managers (and for developers and scientists) it would be therefore of great value to be able to estimate the *effort* of cleaning and integrating some given data sets and to know the pitfalls of such an integration project in advance. This helps deciding about an integration project using cost/benefit analysis, budgeting a team with funds and manpower, and monitoring its progress. Further, knowledge of how well a data source fits into a given data ecosystem improves source selection.

We present an extensible framework for the automatic effort estimation for mapping and cleaning activities in data integration projects with multiple sources. It comprises a set of measures and methods for estimating integration complexity and ultimately effort, taking into account heterogeneities of both schemas and instances and regarding both integration and cleaning operations. Experiments on two real-world scenarios show that our proposal is two to four times more accurate than a current approach in estimating the time duration of an integration process, and provides a meaningful breakdown of the integration problems as well as the required integration activities.

1. COMPLEXITY OF INTEGRATION AND CLEANING

Data integration and data cleaning remain among the most human-work-intensive tasks in data management. Both require a clear understanding of the semantics of schema and data – a notoriously difficult task for machines. Despite much research and development of supporting tools and algorithms, state-of-the-art integration projects involve significant human resource cost. In fact, Gartner reports that 10% of all IT cost goes into enterprise software for data integration and data quality[1,2], and it is well recognized that most of those expenses are for human labor. Thus, when embarking on a data integration and cleaning project, it is useful and important to estimate in advance the effort and cost of the project and to find out which particular difficulties cause these. Such estimations help deciding whether to pursue the project in the first place, planning and scheduling the project using estimates about the duration of integration steps, budgeting in terms of cost or manpower, and finally monitoring the progress of the project. Cost estimates can also help integration service providers, IT consultants, and IT tool vendors to generate better price quotes for integration customers. Further, automatically generated knowledge of how well and how easy a data source fits into a given data ecosystem improves source selection.

However, "*project estimation for [. . .] data integration projects is especially difficult, given the number of stakeholders involved across the organization as well as the unknowns of data complexity and quality.*" [14]. Any integration project has several steps and tasks, including requirements analysis, selection of data sources, determining the appropriate target database, data transformation specifications, testing, deployment, and maintenance. In this paper, we focus on exploring the database-related steps of *integration* and *cleaning* and automatically estimate their effort.

1.1 Challenges

There are simple approaches to estimate in isolation the complexity of individual mapping and cleaning tasks. For the mapping, evaluating its complexity can be done by counting the matchings, i.e., correspondences, among elements. For the cleaning problem, a natural solution is to measure its complexity by counting the number of constraints on the target schema. However, as several integration approaches have shown, the interactive nature of these two problems is particularly complex [5, 11, 13]. For example, a data exchange problem takes as input two relational schemas, a transformation between them (a mapping), a set of target constraints, and answers two questions: whether it is possible to compute a valid solution for a given setting and how. Interestingly, to have a solution, certain conditions must hold on the target constraints, and extending the setting to more complex languages or data models bring tighter restrictions on the class of tractable cases [6, 12].

In our work, the main challenge is to estimate complexity and effort in a setting that goes beyond these ad-hoc studies while satisfying four main requirements:

Generality: We require independence from the language used to express the data transformation. Furthermore, real cases often fail the existence of solution tests considered in formal frameworks,

[1] *http://www.gartner.com/technology/research/it-spending-forecast/*
[2] *http://www.gartner.com/newsroom/id/2292815*

10.5441/002/edbt.2015.07

e.g., weak acyclicity condition [11], but an automatic estimation is still desirable for them in practice.

Completeness: Only a subset of the constraints that hold on the data are specified over the schema. In fact, business rules are commonly enforced at the application level and are not reflected in the metadata of the schemas, but should nevertheless be considered.

Granularity: Details about the integration issues are crucial for consumption of the estimation. For a real understanding and proper planning, it is important to know which source and/or target attributes are cause of problems and how, e.g., phone attributes in source and target schema have different formats. Existing estimators do not reason over actual data structures and thus make no statements about the causes of integration effort.

Configurability and extensibility: The actual effort depends on subjective factors, such as the capabilities of available tools and the desired quality of the output. Therefore, intuitive, yet rich configuration settings for the estimation process are crucial for its applicability. Moreover, users must be able to extend the range of problems covered by the framework.

These challenges cannot be tackled with existing syntactical methods to test the existence of solutions, as they work only in specific settings (*Generality*), are restricted to declarative specifications over the schemas (*Completeness*), and do not provide details about the actual problems (*Granularity*). On the other hand, as systems that compute solutions require human interaction to finalize the process [8, 13], they cannot be used for estimation purpose and their availability is orthogonal to our problem (*Configurability*).

1.2 Approaching Effort Estimation

Figure 1 presents our view on the problem of estimating the effort of data integration. The starting point is an integration scenario with a target database and one or more source databases. The right-hand side of the figure shows the actual integration process performed by an integration specialist, where the goal is to move all instances of the source databases into the target database. Typically, a set of integration tools are used by the specialist. These tools have access to the source and target and support her in the tasks. The process takes a certain effort, which can be measured, for instance as amount of work in hours or days or in a monetary unit.

Our goal is to find that effort without actually performing the integration. Moreover, we want to find and present the problems that cause this effort. To this end, we developed a two-phase process as shown on the left-hand side of Figure 1.

The first phase, the *complexity assessment*, reveals concrete integration challenges for the scenario. To address *generality*, these problems are exclusively determined by the source and target schemas and instances; if and how an integration practitioner deals with them is not addressed at this point. Thus, this first phase is independent of external parameters, such as the level of expertise of the specialist or the available integration tools. However, it is aided by the results of schema matching and data profiling tools, which analyze the participating databases and produce metadata about them (to achieve *completeness*). The output of the complexity assessment is a set of clearly defined problems, such as number of violations for a constraint or number of different value representations. This detailed breakdown of the problems achieves *granularity* and is useful for several tasks, even if not interpreted as an input to calculate actual effort. Examples of application are *source selection* [9], i.e., given a set of integration candidates, find the source with the best 'fit'; and support for *data visualization* [7], i.e., highlight parts of the schemas that are hard to integrate.

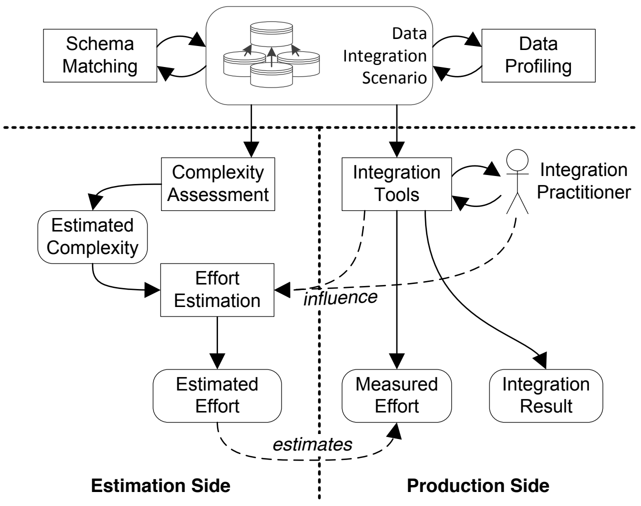


Figure 1: Overview of effort estimation and execution of data integration scenarios.

The second phase, *effort estimation*, builds upon the complexity assessment to estimate the actual effort for overcoming the previously revealed integration challenges in some useful unit, such as workdays or monetary cost. Thereby, this phase addresses *configurability* by taking external parameters into account, such as the experience of the integration practitioner and the features of the integration tools to be used.

1.3 Contributions and structure

Section 2 presents related work and shows that we are the first to systematically address a dimension of data integration and cleaning that has been passed over by the database community but is relevant to practitioners. In particular, we make the following contributions:

- Section 3 introduces the extensible Effort Estimation framework (EFES), which defines a two-dimensional modularization of the estimation problem.
- Section 4 describes an estimation module for structural conflicts between source and target data. This module incorporates a new formalism to compare schemas in terms of mappings and constraints.
- Section 5 reports an estimation module for value heterogeneities that captures formatting problems and anomalies in data that may be missed by structural conflicts.

These building blocks have been evaluated together in an experimental study on two real-world datasets and Section 6 reports on the results. Finally, we conclude our insights in Section 7.

2. RELATED WORK

When surveying computer science literature, a pattern becomes apparent: much technology claims (and experimentally shows) to reduce human effort. The veracity of this claim is evident – after all, any kind of automation of tedious tasks is usually helpful. While for scientific papers this reduction is enough of a claim, the absolute measures of effort and its reduction are rarely explained and measured.

General effort estimation. There are several approaches for effort estimation in different fields, however, none of them considers information coming from the datasets.

In the software domain, an established model to estimate the cost of developing applications is COCOMO [3, 4], which is based on parameters provided by the users such as the number of lines of existing code. Another approach decomposes an overall work task into a smaller set of tasks in a "work breakdown structure" [16]. The authors manually label business requirements with an effort class of simple, medium, or complex, and multiply each of them by the number of times the task must be executed.

In the ETL context, Harden [14] breaks down a project into various subtasks, including requirements, design, testing, data stewardship, production deployment, but also the actual development of the data transformations. For the latter he uses the number of source attributes and assigns for each attribute a weighted set of tasks (Table 1). In sum, he calculates slightly more than 8 hours of work for each source attribute.

Task	Hours per attribute
Requirements and Mapping	2.0
High Level Design	0.1
Technical Design	0.5
Data Modeling	1.0
Development and Unit Testing	1.0
System Test	0.5
User Acceptance Testing	0.25
Production Support	0.2
Tech Lead Support	0.5
Project Management Support	0.5
Product Owner Support	0.5
Subject Matter Expert	0.5
Data Steward Support	0.5

Table 1: Tasks and effort per attribute from [14].

One can find other lists of criteria to be taken into account when estimating the effort of an integration project[3]. These include factors we include in our complexity model, such as number of different sources and types, duplicates, schema constraints, and others we exclude for sake of space from our discussion, such as project management, deployment needs, and auditing. There are also mentions of factors that influence our effort model, such as familiarity with the source database, skill levels, and tool availability. However, merely providing a list of factors is only a first step, whereas we provide novel measures for the database-specific causes for complexity and effort. In fact, existing methods: (i) lack a direct numerical analysis of the schemas and datasets involved in the transformation and cleaning; (ii) do not regard the properties of the datasets at a fine grain and cannot capture the nature of the possible problems in the scenario, (iii) do not consider the interaction of the mapping and the cleaning problems.

Schema-matching for effort estimation. In our work, we exploit schema matching to bootstrap the process. This is along the lines of what authors of matchers suggested. For example, in [24] the authors have pointed out the multiple usages of schema matching tools beyond the concrete generation of correspondences for schema mappings. In particular, they mention "project planning" and "determining the level of effort (and corresponding cost) needed for an integration project". In a similar fashion, in the evaluation of the similarity flooding algorithm, Melnik et al. propose a novel measure "to estimate how much effort it costs the user to modify the proposed match result into the intended result" in terms of additions and deletions of matching attribute pairs [19].

Data-oriented effort estimation. In [25], the authors offer a "back of the envelope" calculation on the number of comparisons needed to be performed by human workers to detect duplicates in a dataset. According to them, the estimate depends on the way the potential duplicates are presented, in particular their order and their grouping. Their ideas fit well into our effort model and show that specific tooling indeed changes the needed effort, independently of the complexity of the problem itself. Complementary work on source selection has focused on the benefit of integrating a new source based on its marginal gain [9, 23].

Data-cleaning and data-transformation. Many systems (e.g., [8, 13]) address the problem of detecting violations over the data given a set of constraints, as we also do in one of our modules for complexity estimation. The challenge for these systems is mostly the automatic repair step, i.e., how to update the data to make it consistent wrt. the given constraints with a minimal number of changes. None of these systems provide tools to estimate the complexity of the repair nor the user effort before actually executing the methods to solve the integration problem. In fact, the challenge is that solving the problem involves the users, and estimating this effort (even in presence of these tools) is our main goal. Similar problems apply to data exchange and data transformation [1, 15].

In the field of model management, the use of metamodels has been investigated to represent in a more general language several alternative data models [2, 21]. Our cardinality-constrained schema graphs (Section 4) can be seen as a proposal for a metamodel with a novel static analysis of cardinalities to identify problems in the underlying schemas and the mapping between them.

3. THE EFFORT ESTIMATION FRAMEWORK

Real-world data integration scenarios host a large number of different challenges that must be overcome. Problems arise in common activities, such as the definition of a mapping between different schemas, the restructuring of the data, and the reconciliation of their value format. We first describe these problems and then introduce our solution.

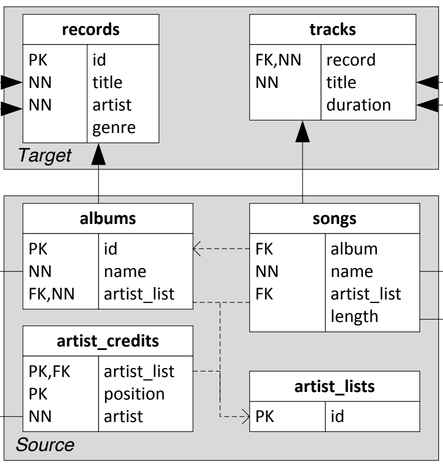
3.1 Data Integration Scenario

A *data integration scenario* comprises: (i) a set of source databases; (ii) a target database, into which the source databases shall be integrated; and (iii) correspondences to describe how these sources relate to the target. Each *source database* consists of a relational schema, an instance of this schema, and a set of constraints, which must be satisfied by that instance. Likewise, the *target database* can carry constraints and possibly already contains data as well that satisfies these constraints. Furthermore, each *correspondence* connects a source schema element with the target schema element, into which its contents should be integrated.

Oftentimes constraints are not enforced at the schema level but rather at the application level or simply in the mind of the integration expert. Even worse, for some sources (e.g., data dumps), a schema definition may be completely missing. To achieve *completeness*, techniques for schema reverse engineering and data profiling [20] can reconstruct missing schema descriptions and constraints from the data.

Example 3.1. *Figure 2 shows an integration scenario with music records data. Both source and target relational schemas (Figure 2a) define a set of constraints, such as primary keys (e.g.,* id *in*

[3]Such as *http://www.datamigrationpro.com/data-migration-articles/2012/2/9/data-migration-effort-estimation-practical-techniques-from-r.html* and *http://www.information-management.com/news/1093630-1.html*



(a) Schemas, constraints, and correspondences.

record	title	duration
1	"Sweet Home Alabama"	"4:43"
1	"I Need You"	"6:55"
1	"Don't Ask Me No Questions"	"3:26"
...		

(b) Example instances from the target table tracks.

album	name	artist_list	length
s3	"Hands Up"	a1	215900
s3	"Labor Day"	a1	238100
s3	"Anxiety"	a2	218200
...			

(c) Example instances from the source table songs.

Figure 2: An example data integration scenario.

records*), foreign keys (*record *in* tracks*, represented with dashed arrows), and not nullable values (*title *in* tracks*).*

Solid arrows between attributes and relations represent correspondences, i.e., two attributes that store the same atomic information or two relations that store the same kind of instances. The source relation albums *corresponds to the target relation* records *and its source attribute* name *corresponds to the* title *attribute in the target. That means, that the albums from the source shall be integrated as records into the target, while the source album names serve as titles for the integrated records.* ◇

We assume correspondences between the source and target schemas to be given, as they can be automatically discovered with schema matching tools [10]. Notice that correspondences are not an executable representation of a transformation, thus they do not induce a *precise* mapping between sources and the target. However, they contain enough information to reason over the complexity of data integration scenarios and detect their integration challenges, as in the following example.

Example 3.2. *The target schema requires exactly one artist value per record, whereas the source schema can associate an arbitrary number of artist credits to each album. This situation implies that integrating any source album with zero artist credits violates the not-null constraint on the target attribute* records.artist*. Moreover, two or more artist credits for a single source album cannot be naturally stored by the single target attribute. Integration practitioners have to solve these conflicts. Hence, this schema heterogeneity increases the necessary effort to achieve the integration.* ◇

Not all kinds of integration issues can be detected by analyzing the schemas, though. The data itself is equally important. While we assume that every instance is valid wrt. its schema, when data is integrated new problems can arise. For example, all sources might be free of duplicates, but there still might be target duplicates when they are combined [22]. These conflicts can also arise between source data and pre-existing target data.

Example 3.3. *Tables 2b and 2c report sample instances of the* tracks *table and the* songs *table, respectively. The duration of tracks in the target database is encoded as a string with the format m:ss, while the length of songs is measured in milliseconds in the source. The two formats are locally consistent, but the source values need a proper transformation when integrated into the target column, thereby demanding a certain amount of effort.* ◇

3.2 A General Framework

Facing different kinds of data integration and cleaning actions, there is the need of different specialized models to decode their complexity and estimate their effort properly. We tackle this problem with our general effort estimation framework EFES. It handles different kinds of integration challenges by accepting a dedicated *estimation module* to cope with each of them independently. Such modularity makes it easier to revise and refine individual modules and establishes the desired *extensibility* by plugging new ones. In this work, we present modules for the three general and, in our experience, most common classes of integration activities: writing an executable mapping, resolving structural conflicts, and eliminating value heterogeneities. While the latter two are explained in subsequent sections, we present the *mapping module* in this section to explain our framework design. For *generality*, the modules do not depend on a fixed language to express the transformations.

Figure 3 depicts the general architecture of EFES. The architecture implements our goal of delivering a set of integration problems and an effort estimate by explicitly dividing the estimation process into an objective *complexity assessment*, which is based on properties of schemas and data, followed by the context-dependent *effort estimation*. We now describe these two phases in more detail.

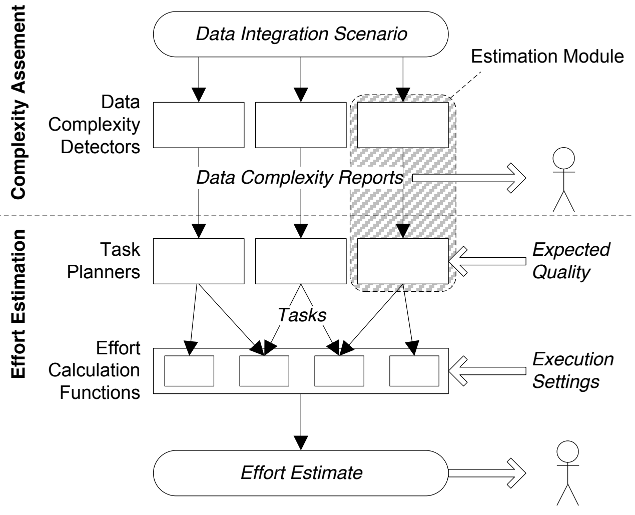


Figure 3: Architecture of EFES.

3.3 Complexity assessment

The goal of this first phase is to compute *data complexity reports* for the integration scenario. These reports serve as the basis for the subsequent effort estimation but also are used to inform the user about integration problems within the scenario. This is particularly useful for source selection [9] and data visualization [7].

Each estimation module provides a *data complexity detector* that extracts complexity indicators from the given scenario and writes them into its report. There is no formal definition for such a report; rather, it can be tailored to the specific, needed complexity indicators. For example, the mapping module builds on the following idea: For each table in the target schema and each source database that provides data for that table, some connection has to be established to fetch the source data and write it into the target table. The overall complexity of the mapping creation is composed of the individual complexities for establishing each of these connections. Furthermore, every connection can be described in terms of certain metrics, such as the number of source tables to be queried, the number of attributes that must be copied, and whether new IDs for a primary key need to be generated.

Example 3.4. *The data complexity report for the scenario in Figure 2 can be found in Table 2. To fetch the data for the* records *table, the three source tables* albums*,* artist_lists*, and* artist_credits *have to be combined, two attributes must be copied, and unique* id *values for the integrated tuples must be generated.* ◇

Target table	Source tables	Attributes	Primary key
records	3	2	yes
tracks	3	2	no

Table 2: Mapping complexity report of the scenario in Figure 2.

3.4 Effort estimation

Based on the data complexity, the effort estimation shall produce a single estimate of the human work to address the different complexities. However, going from an objective complexity measure to a subjective estimate of human work requires external information about the context. We distinguish one aspect that is specific to the data integration problem, (i) the *expected quality* of the integration result, and, as a more common aspect, (ii) the *execution settings* for the scenario.

(i) Expected quality: Data cleaning is the operation of updating an instance such that it becomes consistent wrt. any constraint defined on its schema [8, 13]. However, such results can be obtained by automatically removing problematic tuples, or by manually solving inconsistencies involving a domain expert. Each choice implies different effort.

Example 3.5. *Consider again Example 3.3 with the duration format mismatch. As* duration *is nullable, one simple way to solve the problem is to drop the values coming from the new source. A better, higher quality solution is to transform the values to a common format, but this operation requires a script and a validation by the user, i.e., more effort [15].* ◇

(ii) Execution settings: The execution settings represent the circumstances under which the data integration shall be conducted. Examples of common context information are the expertise of the integration practitioners and their familarity with the data [4]. In our setting, we also model the level of automation of the available integration tools, and how critical the errors are, e.g., integrating medical prescriptions requires more attention (and therefore effort) than integrating music tracks.

Example 3.6. *Consider again the problem with the cardinality of record artists. There are* schema mappings *tools [18] that are able to automatically create a synthetic value in the target, if a source album has zero artist credits, and to automatically create multiple occurrences of the same album with different artists, if multiple artists are credited for a single source album. Such tools would reduce the mapping effort.* ◇

For the effort estimation, each estimation module has to provide a *task planner* that consumes its data complexity report and outputs *tasks* to overcome the reported issues. Each of these tasks is of a certain type, is expected to deliver a certain result quality, and comprises an arbitrary set of parameters, such as on how many tuples it has to be executed. We defined two instances of expected quality, namely *low effort* (removal of tuples) and *high quality* (updates). This criterion is extensible to other repair actions, but it already allows to choose between alternative cleaning tasks as shown in Example 3.5.

Example 3.7. *A complexity report for the scenario from Figure 2 states that there are albums without any artist in the source data that lead to a violation of a not-null constraint in the target. The corresponding task model proposes the alternative actions* Reject violating tuple *(low effort) or* Add missing value *(high quality) to solve the problem.* ◇

Once the list of tasks has been determined, the effort for their execution is computed. For this purpose, the user specifies in advance for each task type an *effort-calculation function* that can incorporate task parameters. As an example, we report the effort-calculation functions for the execution settings of our experiments in Table 9. The framework uses these functions to estimate the effort for each of the tasks. Finally, the total of all these task estimates forms the overall effort estimate.

Example 3.8. *We exemplify the effort-calculation functions for the tasks derived from the report in Table 2. The* Create mapping *task might be done manually with SQL queries. Then an adequate function would be*

$$\textit{effort} = 3 mins \cdot \textit{tables} + 1 min \cdot \textit{attributes} + 3 mins \cdot \textit{PKs}$$

leading to an overall effort of 25 (18 + 4 + 3) minutes. However, if a tool can generate this mapping automatically based on the correspondences (e.g., [18]), then a constant value, such as $\textit{effort} = 2 mins$*, can reflect this circumstance, leading to an overall effort of four minutes.* ◇

The above described task-based approach offers several advantages over an immediate complexity-effort mapping [14], where a formula directly converts statistics over the schemas into an effort estimation value. Our model enables *configurability*, as it treats execution settings as a first-class component in the effort-calculation functions and these can be arbitrarily complex as needed. Furthermore, instead of just delivering a final effort value, our effort estimate is broken down according to its underlying tasks. This *granularity* helps users understand the required work and explains how the estimate has been created, thus giving the users the opportunity to properly plan the integration process.

4. STRUCTURAL CONFLICTS

Structural heterogeneities between source and target data structures are a common problem in integration scenarios. This section describes a module to detect these problems and estimate the effort arising out of them. It can be plugged into the framework architecture in Figure 3 with the following workflow: Its data complexity detector (*structure conflict detector*) analyzes how source and target data relate to each other, and counts the number of emerging structural conflicts. Based on those conflicts, the task planner (*structure repair planner*) then (i) derives a set of cleaning tasks to make the conflicting source data fit into the target schema, and (ii) estimates how often each such task has to be performed. These tasks can finally be fed into the effort calculation functions.

4.1 Structure Conflict Detector

In the first step of structural conflict handling, all source and target schemas of the given scenario are converted into *cardinality-constrained schema graphs* (short *CSG*), a novel modeling formalism that we specifically devised for our task. It offers a single, yet expressive constraint formalism with a set of inference operators that allow elegant comparisons of schemas. Additionally, it is more general than the relational model and can describe (integrated) database instances that do not conform to the relational model. For instance, an integrated tuple might provide multiple values for a single attribute, like in Example 3.2. The higher expressiveness of CSGs allows to reason about necessary cleaning tasks to make the integrated databases conform to the relational model. In the following, we formally define CSGs and explain how to convert relational databases into CSGs.

DEFINITION 1. *A CSG is a tuple $\Gamma = (N, P, \kappa)$, where N is a set of nodes and $P \subset N^2$ is a set of relationships. Furthermore, $\kappa\colon P \rightarrow 2^{\mathbb{N}}$ expresses schema constraints by prescribing cardinalities for relationships.*

DEFINITION 2. *A CSG instance is a tuple $\mathcal{I}(\Gamma) = (\mathcal{I}_N, \mathcal{I}_P)$, where $\mathcal{I}_N$ assigns a set of elements to each node in N and $\mathcal{I}_P$ assigns to each relationship links between those elements.*

To convert a relational schema, for each of its relations, a corresponding *table node* (rectangle) is created to represent the existence of tuples in that relation. Furthermore, for each attribute, an *attribute node* (round shape) is created and connected to its respective table node via a *relationship*. While these attribute nodes hold the *set of distinct* values of the original relational attribute, the relationships link tuples and their respective attribute values. With this proceeding, any relational database can be turned into a CSG without loss of information.

Example 4.1. *Figure 4 depicts two CSGs for the example scenario schemas in Figure 2a, one for the source and one for the target schema*[4]. *For instance, the example* tracks *tuple t = (1, "Sweet Home Alabama", "4:43") from Figure 2b is represented in the CSG instance as follows: The table node* tracks $\in N$ *holds an abstract element id_t, i.e., $id_t \in \mathcal{I}_N(\text{tracks})$, representing the tuple's identity. Likewise,* record $\in N$ *holds exactly once the value 1, i.e., $1 \in \mathcal{I}_N(\text{record})$, and the relationship $\rho_{\text{tracks}\rightarrow\text{record}}$ contains a* link *for these elements, i.e., $(id_t, 1) \in \mathcal{I}_P(\rho_{\text{tracks}\rightarrow\text{record}})$, thus stating that $t[\text{record}] = 1$. The other values for the* title *and* duration *attributes are represented accordingly. Furthermore, foreign key relationships are represented by special* equality relationships *(dashed line) that link all equal elements of two nodes, e.g., all common values of the* id *and* record *nodes in the target CSG of Figure 4.* ◇

[4]Some correspondences between the schemas are omitted for clarity, but are not generally discarded.

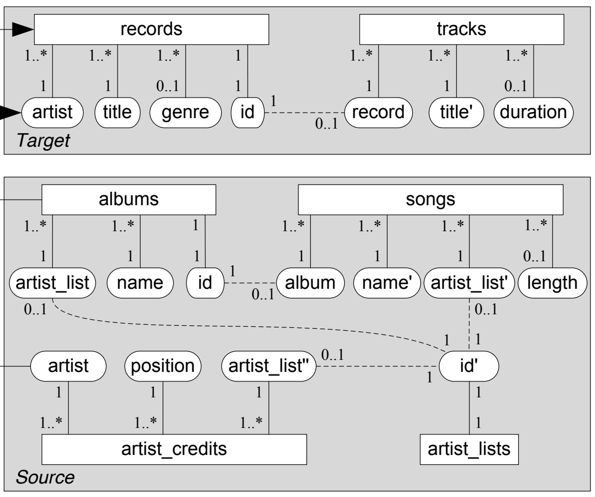


Figure 4: The integration scenario translated into cardinality-constrained schema graphs.

To express schema constraints in CSGs, all relationships are annotated with *prescribed cardinalities*, that restrict the number of elements and/or values of connected nodes that must relate to each other via the annotated relationship. For example, tracks.record is not nullable, which means, that each tracks tuple must provide exactly one record value. Translated to CSGs, this means that for each tuple t_i, the relationship $\rho_{\text{tracks}\rightarrow\text{record}}$ must contain exactly one link:

$$\forall t_i\colon\ |\{v \in \mathcal{I}_N(\text{record}) \mid (id_{t_i}, v) \in \mathcal{I}_P(\rho_{\text{tracks}\rightarrow\text{record}})\}| = 1 \quad .$$

Formally, this is expressed by $\kappa(\rho_{\text{tracks}\rightarrow\text{record}}) = \{1\}$, which is also graphically annotated in Figure 4. However, tracks.record is not subject to a unique-constraint. In consequence, every record value can be found in one or more tuples. Therefore, $\kappa(\rho_{\text{record}\rightarrow\text{tracks}}) = 1..* = \{1, 2, 3, \ldots\}$. By means of prescribed cardinalities, unique, not-null, and foreign key constraints can be expressed, as well as two conformity rules for relational schemas: each tuple can have at most one value per attribute, and each attribute value must be contained in a tuple.

As stated above, another important feature of CSGs is the ability to combine relationships into *complex relationships* and to analyze their properties. As one effect, prescribing cardinalities not only to atomic but also to complex relationships further allows to express n-ary versions of the above constraints and functional dependencies. We devised the following relationship construction operators:

'$\circ$': The *composition* concatenates two adjacent relationships. Formally, $\mathcal{I}_P(\rho_1 \circ \rho_2) \overset{def}{=} \mathcal{I}_P(\rho_1) \circ \mathcal{I}_P(\rho_2)$.

'$\cup$': The *union* of two relationships $\rho_1 \cup \rho_2$ contains all links of the two relationships, i.e., $\mathcal{I}_P(\rho_1 \cup \rho_2) \overset{def}{=} \mathcal{I}_P(\rho_1) \cup \mathcal{I}_P(\rho_2)$. This is particularly useful, when multiple source relationships need to be combined.

'$\bowtie$': The *join* operator connects links from relationships $\rho_{A\rightarrow C}$, $\rho_{B\rightarrow C}$ with equal codomain values, thereby inducing a relationship between $A \times B$ and C. Formally, $\mathcal{I}_P(\rho_{A\rightarrow C} \bowtie \rho_{B\rightarrow C}) \overset{def}{=} \{((a, b), c) : (a, c) \in \mathcal{I}_P(\rho_{A\rightarrow C}) \wedge (b, c) \in \mathcal{I}_P(\rho_{A\rightarrow B})\}$. The join can be combined with other operators to express n-ary uniqueness constraints.

'$\|$': The *collateral* of two relationships $\rho_{A\to B}\|\rho_{C\to D}$ induces a relationship between $A \times C$ and $B \times D$: $\mathcal{I}_P(\rho_{A\to B}\|\rho_{C\to D}) \overset{def}{=} \{((a,c),(b,d)) : (a,b) \in \mathcal{I}_P(\rho_{A\to B}) \wedge (c,d) \in \mathcal{I}_P(\rho_{C\to D})\}$. The collateral can be applied to express n-ary foreign keys.

Based on these definitions, efficient algorithms can be devised to infer the constraints of complex relationships.

LEMMA 1. *Let $\rho_1, \rho_2 \in P$ be two relationships in a graph Γ and ρ_1's end node is ρ_2's start node. Then the cardinality κ of $\rho_1 \circ \rho_2$ can be inferred as*

$$\begin{aligned} &\kappa(\rho_1 \circ \rho_2) \overset{def}{=} \kappa(\rho_1) \circ \kappa(\rho_2) \\ = \;& a_1..b_1 \circ a_2..b_2 \overset{def}{=} (\operatorname{sgn} a_1 \cdot a_2)..(b_1 \cdot b_2) \end{aligned}$$

where $\operatorname{sgn}(0) = 0$ *and* $\operatorname{sgn}(n) = 1$ *for* $n > 0$.

LEMMA 2. *Let $\rho_1, \rho_2 \in P$ be two relationships in a graph Γ. Then the cardinality of $\rho_1 \cup \rho_2$ can be inferred as*

$$\kappa(\rho_1 \cup \rho_2) \overset{def}{=} \begin{cases} \kappa(\rho_1) \cup \kappa(\rho_2) & \textit{if } \mathcal{I}_P(\rho_1) \textit{ and } \mathcal{I}_P(\rho_2) \textit{ have disjoint domains} \\ \kappa(\rho_1) + \kappa(\rho_2) & \textit{if } \mathcal{I}_P(\rho_1) \textit{ and } \mathcal{I}_P(\rho_2) \textit{ have equal domains but disjoint codomains} \\ \kappa(\rho_1) \hat{+} \kappa(\rho_2) & \textit{if } \mathcal{I}_P(\rho_1) \textit{ and } \mathcal{I}_P(\rho_2) \textit{ have equal domains and overlapping codomains} \end{cases}$$

where $\kappa_1 + \kappa_2 \overset{def}{=} \{a + b : a \in \kappa_1 \wedge b \in \kappa_2\}$ *and* $\kappa_1 \hat{+} \kappa_2 \overset{def}{=} \{c : a \in \kappa_1 \wedge b \in \kappa_2 \wedge \max\{a, b\} \leq c \leq (a + b)\}$.

Note, that Lemma 2 can also be applied to relationships with partially overlapping domains by splitting those into the overlapping and the disjoint parts.

LEMMA 3. *Let $\rho_1, \rho_2 \in P$ be two relationships in a graph Γ with a common end node and let $m = \min\{\max \kappa(\rho_1), \max \kappa(\rho_2)\}$. Then the cardinality of $\rho_1 \bowtie \rho_2$ can be inferred as*

$$\kappa(\rho_1 \bowtie \rho_2) \overset{def}{=} \begin{cases} \emptyset & \textit{if } m = 0 \vee m = \perp \\ 1..m & \textit{otherwise} \end{cases}$$

and its inverse cardinality as

$$\begin{aligned} &\kappa((\rho_1 \bowtie \rho_2)^{-1}) \\ &\overset{def}{=} (\min \kappa(\rho_1) \cdot \min \kappa(\rho_2))..(\max \kappa(\rho_1) \cdot \max \kappa(\rho_2)) \end{aligned}$$

LEMMA 4. *Let $\rho_1, \rho_2 \in P$ be two relationships in a graph Γ. Then the cardinality of $\rho_1 \| \rho_2$ can be inferred as*

$$\kappa(\rho_1 \| \rho_2) \overset{def}{=} 0..(\max \kappa(\rho_1) \cdot \max \kappa(\rho_2))$$

Given the means to combine relationships and infer their cardinality, it is now possible to compare the structure of source and target schemas. As data integration seeks to populate the target relationships with data from the sources, the structure conflict detector must determine how the atomic target relationships are represented in the source schemas. In general, target relationships can correspond to arbitrarily complex source relationships, in particular to compositions. The composition operator particularly allows to treat the matching of target relationships to source relationships as a graph search problem, as is exemplified with the atomic target relationship records $\to$ artist from Figure 4.

First, the relationship's start and end node are matched to nodes in the source schema via the correspondences, in this case to albums and artist. Then, a path is sought between those nodes. In the example, there are two possible paths, namely albums $\to$ artist_list $\to$ id$'$ $\to$ artist_list$''$ $\to$ artist_credits $\to$ artist, and albums $\to$ id $\to$ album $\to$ songs $\to$ artist_list$'$ $\to$ id$'$ $\to$ artist_list$''$ $\to$ artist_credits $\to$ artist. To resolve this ambiguity, it is assumed that the most *concise* detected source relationship is the best match for the atomic target relationship. A relationship is more concise than another relationship, if its (inferred) cardinality κ_1 is more specific than the other relationship's cardinality κ_2, i.e., $\kappa_1 \subset \kappa_2$. In the case of equal cardinalities, the shorter relationship is preferred, according to Occam's razor principle[5]. Here, both detected relationships have the same inferred cardinality $0..*$ according to Lemma 1, but the former is shorter and therefore selected as match.

Having matched a target relationship to a source relationship, comparing these two can finally reveal structural conflicts. The example target relationship records $\to$ artist has the annotated cardinality 1, but its corresponding source relationship is less concise, having an inferred cardinality of $0..*$. This lower conciseness causes a structural conflict: The target schema accepts only one artist value per record, while the source potentially offers an arbitrary amount of artists per album. To refine the statement about this violation, we can count the number of albums in the source data, that are associated to no or more than one artist, hence, determining the number of actually conflicting data elements. This *violation count* is applicable to any database constraint that can be expressed in CSG as listed above. Supporting more advanced constraints in CSGs, such as *conditional functional dependencies* [8], is left for future work.

The above described matching and checking process is performed for each target relationship. In the example scenario, there is only one more structural violation: artist $\to$ records has $0..*$ as inferred cardinality, so there may be artists with no albums. Afterwards, all collected structure violations, depicted in Table 3, are forwarded to the structure repair planner.

Constraint in target schema	Violation count in source data
$\kappa(\rho_{\text{records}\to\text{artist}}) = 1$	503
$\kappa(\rho_{\text{artist}\to\text{records}}) = 1..*$	102

Table 3: Complexity report of the structure conflict detector.

4.2 Structure Repair Planner

The structure repair planner proposes necessary cleaning tasks to cope with the structural violations in an integration scenario, that form the base for the following effort calculation. It ships with ten such cleaning tasks listed in Table 4; one per type of violation, e.g., of a not-null constraint, and expected result quality (low or high). The structure conflict detector can automatically select exactly those tasks that the integration practitioner has to perform in the data integration scenario to fix structural violations.

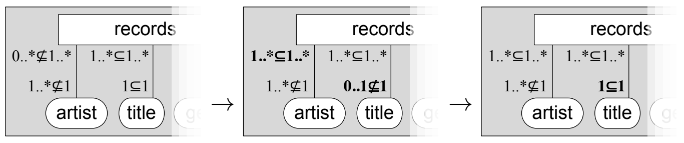
However, simply designating a task for each given violation is not sufficient, as data cleaning operations usually have side effects that can cause new violations. For instance, the structure conflict detector reveals that there are 102 artists in the source data that have no albums and can thus not be represented in the target schema.

[5]Among competing hypotheses, the one with the fewest assumptions should be selected.

	Result quality	
Constraint	Low effort	High quality
Not null violated	Reject tuple	Add missing value
Unique violated	Set values to null	Aggregate tuples
Multiple attribute values	Keep any value	Merge values
Value w/o enclosing tuple	Drop value	Create enclosing tuple
FK violated	Delete dangling value	Add referenced value

Table 4: Structural conflicts and their corresponding cleaning tasks.

The high-quality solution is to apply the task *Create tuples for detached values*, that creates record tuples to store these artists, so that they do not have to be discarded. These new tuples would violate the not-null constraint on the title attribute, though, so subsequent cleaning tasks are necessary. To account for such impacts, we simulate applied cleaning tasks on *virtual CSG instances* as exemplified in Figure 5. In addition to the prescribed cardinalities, the target CSG is annotated with *actual cardinalities*. In contrast to the prescribed cardinalities, those do not *prescribe* schema constraints but *describe* the state of the (conceptually) integrated source data – in terms of its relationships' cardinalities. Hence, the actual cardinalities are initialized with the inferred cardinalities from the source database. Figure 5a depicts this initial state. As long as there are actual cardinalities (on the left-hand side) that are not subsets of the prescribed ones, the CSG instance is invalid wrt. its constraints. Now, if the structure repair planner has chosen a cleaning task, e.g., adding new records tuples for artists without albums, its (side) effects are simulated by modifying the actual cardinalities, as shown in Figure 5b with bold print. So, amongst others the actual cardinality of artist $\rightarrow$ records is changed from $0..*$ to $1..*$, reflecting that all artists appear in a record after the task, and the cardinality of records $\rightarrow$ title is altered from 1 to 0..1, stating that some records would then have no title. The latter forms a new constraint violation. Now, a successive repair task can be applied on this altered CSG instance, e.g., the task *Add missing values*, which leads to the state of Figure 5c.



(a) Initial state. (b) State after *Add new tuples* for records. (c) State after *Add missing values* for title.

Figure 5: Extract of a virtual CSG instance as cleaning tasks are performed on it.

This procedure of picking a task and simulating its effects is repeated until the virtual CSG instance contains no more violations. Furthermore, the structure repair planner orders the repair tasks, so that tasks that cause new structural violations (or might break an already fixed violation) precede the task that fixes this violation. This is not computationally expensive, because we need to order only tasks that affect a common relationship, but doing so allows for the detection of "infinite cleaning loops", where the execution order of cleaning tasks forms a cycle. In most cases, these cycles are a consequence of contradicting repair tasks. EFES proposes only consistent repair strategies. Additionally, the knowledge of the necessary cleaning tasks in a data integration scenario, including their order, are a valuable aid that can positively impact the integration effort spent on coping with structural conflicts. Therefore, the ordered task list is provided to the user. Finally, the determined cleaning tasks are fed into the user-defined effort calculation functions, which automatically determine the effort for dealing with structural violations in the given scenario. Table 5 presents this effort for the example scenario.

Task	Repetitions	Effort
Add tuples (records)	102	5 mins
Add missing values (title)	102	204 mins
Merge values (title)	503	15 mins
Total		224 mins

Table 5: High-quality structure repair tasks and their estimated effort using the effort calculation functions from Table 9.

5. VALUE HETEROGENEITIES

Value heterogeneities are a frequent class of data integration problems with a common factor: corresponding attributes in the source and target schema use different representations for their values. For instance, in Example 3.3 the target table tracks stores song durations as strings, whereas the source table songs stores these durations in milliseconds as integers. An integration practitioner might therefore want to convert or discard the source values to avoid having different value representations in the tracks.duration attribute. Thus, value heterogeneities can increase the integration effort.

This section presents a module in EFES to estimate the effort caused by value heterogeneities. The data complexity is computed by the *value fit detector*, which analyzes the source and target data to detect different types of value heterogeneities between them. These heterogeneities are then reported to the *value transformation planner*, the task model that proposes data cleaning tasks in response to the heterogeneity issues. Finally, the effort for the proposed tasks can be calculated.

5.1 Value Fit Detector

The basic approach of the value fit detector is to aggregate source and target data into statistics and compare these statistics to detect heterogeneities. Statistics are eligible for this evaluation, because they allow efficient comparison for large amounts of data, while enabling extensibility (as new functions can be added) and completeness (as issues that are not captured by available metadata can be discovered). Furthermore, statistics help to detect the especially meaningful, general data properties that characterize the data as a whole. In particular, if the source data does not match the observed or specified characteristics of the target dataset, plainly integrating this source data would impair the overall quality of the integration result: integration practitioners might want to spend effort to make source data consistent with the target data characteristics.

The value fit detector implements this idea as follows: Given an integration scenario, it processes all pairs of source and target attributes that are connected by a correspondence. For each such pair, statistic values of both attributes are calculated, with the target attribute's datatype designating which exact statistic types to use. In particular, we consider the following statistics:

- The *fill status* counts the null values in an attribute and the values that cannot be cast to the target attribute's datatype.

- The *constancy* is the inverse of Shannon's information entropy and is useful to classify whether the values of an attribute come from a discrete domain [17].
- The *text pattern statistic* collects frequent patterns in a string attribute.
- *Character histogram* captures the relative occurrences of characters in a string attribute.
- The *string length statistic* determines the average string length and its standard deviation for a string attribute.
- Similarly, the *mean statistic* collects the mean value and standard deviation of a numeric attribute.
- The *histogram statistic* describes numeric attributes as histograms.
- *Value ranges* are used to determine the minimum and maximum value of a numeric attribute.
- For attributes with values from a discrete domain, the *top-k values statistic* identifies the most frequent values.

For Example 3.3, the string-typed duration target attribute designates the fill status, the text pattern statistic, the character histogram, the string length statistic, and the top-k values as interesting statistics to be collected.

In the next step, a decision model identifies, based on the gathered statistics values, the different types of value heterogeneities within the inspected attribute pair. Algorithm 1 outlines this decision model, which consists of a sequence of rules. The evaluation of each rule has its own, mostly simple, logic. The first rule ($substantiallyFewerSourceValues$), for instance, is evaluated by comparing the fill status statistics of the source and target attribute.

Algorithm 1: Detect value heterogeneities.

```
Data: source attribute statistics $\mathcal{S}_s$, target attribute statistics $\mathcal{S}_t$
Result: value heterogeneities $V$
1  if substantiallyFewerSourceValues($\mathcal{S}_s, \mathcal{S}_t$) then
2      add Too few source elements to $V$;
3  if hasIncompatibleValues($\mathcal{S}_s$) then
4      add Different value representations (critical) to $V$;
5  if domainRestricted($\mathcal{S}_s$) $\wedge$ $\neg$domainRestricted($\mathcal{S}_t$) then
6      add Too coarse-grained source values to $V$;
7  else if $\neg$domainRestricted($\mathcal{S}_s$) $\wedge$ domainRestricted($\mathcal{S}_t$) then
8      add Too fine-grained source values to $V$;
9  else if domainSpecificDifferences($\mathcal{S}_s, \mathcal{S}_t$) then
10     add Different value representations to $V$;
```

For the above example attribute pair, the fill-statuses are for both attributes near 100 %, there are no incompatible source values (integers can always be cast to strings), and neither of the attributes is domain-restricted. Still, possible domain-specific differences between them might be present. The evaluation of this last rule is more complex. For this purpose, a set of statistics, that are specific to the target attribute's datatype, are computed, e.g., the string format and string length statistic for the string-typed, not domain-restricted duration attribute. To compare these statistics among attributes, for each of them of type τ, an *importance score* $i\big(\mathcal{S}_t(\tau)\big)$ and a *fit value* $f\big(\mathcal{S}_s(\tau), \mathcal{S}_t(\tau)\big)$ are calculated. These calculations are specific to the actual statistics. Intentionally, the importance score describes how important the statistic type at hand is for the target attribute. For example, in the duration attribute, all values have the same text pattern *[number ":" number]*, so the string format statistic is presumably an important characteristic and should therefore have a high importance score. If it had many different text patterns in contrast, its importance would be close to 0. In addition, the fit value measures to what extent the source attribute statistics fit into the target attribute statistics. For instance, the length attribute provides only values with the differing pattern *[number]* leading to a low fit value. Eventually, the fit values for all applied statistic types are averaged using the importance scores as weights:

$$f \overset{def}{=} \sum_{\tau} \Big(i\big(\mathcal{S}_t(\tau)\big) \cdot f\big(\mathcal{S}_s(\tau), \mathcal{S}_t(\tau)\big) \Big)$$

This overall fit value tells to what extent the source attribute fulfills the most important characteristics of the target attribute. If it falls below a certain threshold, we assume domain-specific differences in between the compared attributes and Algorithm 1 issues an according value heterogeneity, e.g., *Different value representations* between the attributes length and duration. In experiments with importance scores and fit values between 0 and 1, we found 0.9 to be a good threshold to separate seamlessly integrating attribute pairs from those that had notably different characteristics.

The set of all value heterogeneities for all attribute pairs forms the complexity report of the value fit detector that can in the following be processed by the value transformation planner. Table 6 shows the complexity report for the example scenario. Note that the value heterogeneities can carry additional information that are derived from the attribute statistics as well and that can be useful to produce accurate estimates. These parameters are not further described in this paper.

Value heterogeneity	**Additional parameters**
Different value representation (length $\rightarrow$ duration)	274.523 source values, 260.923 distinct source values

Table 6: Complexity report of the value fit detector.

5.2 Value Transformation Planner

The value transformation planner proposes tasks to solve value heterogeneities as specified in Table 7. In contrast to the structure repair tasks from Section 4.2, those tasks do not have interdependencies. Therefore, the value transformation planner can simply propose an appropriate task for each given value heterogeneity based on the expected result quality of the data integration. For the four different types of value heterogeneities, there are only five different tasks, because for a low-effort integration result, value heterogeneities can in most cases be simply ignored. So, the *Different value representations* between the duration and length attributes might either be neglected (leading to no additional effort) or, for a high-quality integration result, the value fit detector issues the task *Convert values*. This task is then again fed into the effort calculation functions that compute the effort that is necessary for the task completion. Table 8 illustrates the resulting effort estimate.

6. EXPERIMENTS

To show the viability of the general effort estimation architecture and its different models, we conducted experiments with real-world data from two different domains. In the following, we first introduce the system and its configuration. We then describe the

Value heterogeneity	**Result quality** low effort	high quality
Too few elements	-	Add values
Different representations		
critical	Drop values	Convert values
uncritical	-	Convert values
Too specific	-	Generalize values
Too general	-	Refine values

Table 7: Value heterogeneities and corresponding cleaning tasks.

Task	**Parameters**	**Effort**
Convert values (length → duration)	274.523 values, 260.923 distinct values	15 mins
Total		15 mins

Table 8: Value transformation tasks and their estimated effort.

integration scenarios and how we created the ground truth effort by manually integrating them. Finally, we compare our system to a baseline approach from the literature and to the measured effort in creating the ground truth.

6.1 Setup

The EFES prototype is a Java-based implementation of the effort estimation architecture presented in the paper, along with the three modules discussed. It offers multiple configuration options via an XML file and a command-line interface. As input, the prototype takes correspondences between a source and a target dataset, all stored in PostgreSQL databases. The prototype and the datasets are available for download.[6]

Configuration and External Factors. In our experiments, the only available software for conducting the integration are (i) manually written SQL queries, and (ii) a basic admin tool like pgAdmin. We also assume the user has not seen the datasets before and that she is familiar with SQL. Based on these external factors, corresponding effort conversion functions are reported in Table 9. For example, the intuition behind the formula for adding values is that it takes a practitioner two minutes to investigate and provide a single missing value. In contrast, deleting tuples with missing values requires five minutes, because independently from the number of violating tuples, one can write an appropriate SQL query to perform this task. Furthermore, we fine-tuned these settings for our experiments as we explain in Section 6.2.

Integration Scenarios. We considered two real-world case studies. The first is the well-known Amalgam dataset from the bibliographic domain, which comprises four schemas with between 5 and 27 relations, each with 3 to 16 attributes. The second is a new case study we created with a set of three datasets with discographic data. In those datasets, there are three schemas with between 2 and 56 relations and between 2 and 19 attributes each. Links to the original datasets can be found on the web page mentioned above.

For each case study, we created four integration scenarios, each consisting of a source and target database and manually created correspondences, because we do not want the evaluation results to depend on the quality of a schema matcher at this point. Within each domain, we included a data integration scenario with identical source and target schema and three other, randomly selected

[6] *http://hpi.de/naumann/projects/repeatability/*

Task	**Effort function (mins)**
Aggregate values	$3 \cdot$ #repetitions
Convert values	(if #dist-vals < 120) 30, (else) $0.25 \cdot$ #dist-vals
Generalize values	$0.5 \cdot$ #dist-vals
Refine values	$0.5 \cdot$ #values
Drop values	10
Add values	$2 \cdot$ #values
Create enclosing tuples	10
Delete detached values	0
Reject tuples	5
Keep any value	5
Add tuples	5
Aggregate tuples	5
Delete dangling values	5
Add referenced values	5
Delete dangling tuples	5
Unlink all but one tuple	5
Write mapping	$3 \cdot$ #FKs $+ 3 \cdot$ #PKs $+$ #atts $+ 3 \cdot$ #tables

Table 9: Effort calculation functions used for the experiments.

scenarios with different schemas.

Effort Estimates. In order to obtain effort estimations, we applied the following procedure to each data scenario twice, once striving for a low-effort integration result, once for a high-quality result. At first, we executed EFES on the scenario to obtain the data complexity reports and a set of initially proposed mapping and cleaning tasks. If a data complexity aspect was properly recognized but we preferred a different integration task, we have adapted the proposed tasks. For instance, in one scenario, our prototype proposed to provide missing *FreeDB IDs* for music CDs to obtain a high-quality result; this ID is calculated from the CD structure with a special algorithm. Since there was no way for us to obtain this value, we exchanged this proposal with *Reject tuples* to delete source CDs without such a disc ID instead.

Ground Truth Effort. Finally, we gathered the ground truth of necessary integration tasks manually and conducted them with SQL scripts and *pgAdmin*, thereby measuring the execution time of each task. We believe these two manually integrated scenarios with time annotations are a contribution per se, as they can be used also for benchmarking of mapping and cleaning systems for other data integration projects.

6.2 Experimental Results

The correspondences that were created for the case study datasets have been fed into EFES to compare its effort estimates to the actual effort. As a baseline, we used the standard approach based on attribute counting [14], as discussed in Section 2. To obtain fair calibrations of EFES and this baseline model, we employed cross validation: We used the effort measurements from the bibliographic domain to calibrate the parameters of EFES and the attribute counting approach for the estimation of the music domain scenarios, and vice versa. Thus, we have for both scenarios different training and test data and both models can be regarded as equally well-trained. To compare the two models against the measured effort, we applied the root-mean-square error (rmse):

$$\text{rmse} = \sqrt{\frac{\sum_{s \in S} \left(\frac{\text{measured}(s) - \text{estimated}(s)}{\text{measured}(s)}\right)^2}{\#\text{scenarios}}}$$

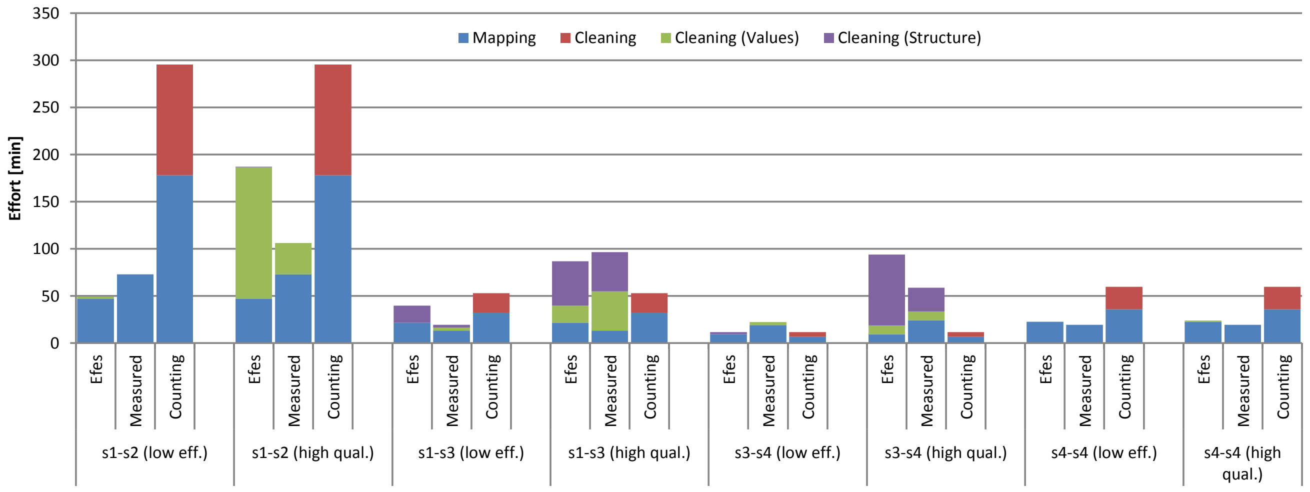


Figure 6: Effort estimates (EFES), actual effort (Measured), and baseline estimates (Counting) of the Bibliographic scenario.

where S is a set of integration scenarios.

We start our analysis with the Amalgam dataset with the results reported in Figure 6. EFES consistently outperforms the counting approach in all scenarios. This is explained by the fact that the baseline has no concept of heterogeneity between values in the datasets, but it is one of the main complexity drivers in these integration scenarios. In terms of the root-mean-square error, EFES achieves 0.47, while the baseline obtains 1.90 (lower values indicates better estimations), thus there is an improvement in the effort estimation by a factor of four. Moreover, EFES not only provides the total number of minutes, but also a detailed break down of where the effort is to be expected. This turned out to be particular useful to revise the effort estimate as described above, thus enriching the estimation process with further input. In fact, it makes a significant difference if an integration practitioner has to add hundreds of missing values or if tuples with missing values are dropped. The baseline approach also distinguishes between mapping and cleaning efforts, but it relates them neither to integration problems nor actual tasks. The *s4-s4* scenario demonstrates this: source and target database have the same schema and similar data, so there are no heterogeneities to deal with. While we can detect this, the counting approach estimates considerable cleaning effort.

When we move to the music datasets, the results in Figure 7 show a smaller difference between the two estimation approaches. In fact, EFES outperforms the baseline four times, in three cases baseline does a better job, and in one case the estimate is basically the same. The explanation is that in this domain, there are fewer problems at the data level and the effort is dominated by the mapping, which strongly depends on the schema. However, when we look at the root-mean-square error, EFES achieves 1.05, while the baseline obtains 1.64. Therefore, even in cases where EFES cannot exploit all of its modules, and when counting should perform at its best, our systematic estimation is better.

It is important to consider the generality of the presented comparison. The two case studies are based on real-world data sets with different complexity and quality. When putting the results over the eight scenarios together, EFES achieves a root-mean-square error of 0.84, while the baseline obtains 1.70. In terms of execution time, EFES relies on simple SQL queries only for the analysis of the data and completes within seconds for databases with thousands of tuples. This overhead can be neglected in the context of the dominating integration cost.

7. CONCLUSIONS

We have tackled the problem of estimating the complexity and the effort for data integration scenarios. As data integration is composed of many different activities, we proposed a novel system, EFES, that integrates different ad-hoc estimation modules in a unified fashion. We have introduced three modules that take care of estimating the complexity and effort of (i) mapping activities, (ii) cleaning of structural conflicts arising because of different structures and integrity constraints, and (iii) resolving heterogeneity in integrated data, such as different formats. Experimental results show that our system outperforms the standard baseline up to a factor of four in terms of precision of the estimated effort time in minutes. When compared to the effective time required by a human to achieve integration, EFES provides a close estimate for most of the cases.

We believe that our work is only a first step in this challenging problem. One possible general direction is to integrate EFES with approaches that measure the benefit of the integration, such as the marginal gain [9]. This integration would allow to plot cost-benefit graphs for the integration: the more effort, the better the quality of the result.

A rather technical challenge in our system is to drop the assumption that correspondences among schemas are given. In practice, the effort for creating quality correspondences cannot be completely neglected – although, in our experience it takes considerably less time than other integration activities – and automatically generated correspondences introduce uncertainty wrt. the produced estimates. The accuracy measure as proposed Melnik et al. [19] seems to be a good starting point to tackle this issue.

Acknowledgments. We would like to thank ElKindi Rezig for valuable discussions in the initial phase of this project.

8. REFERENCES

[1] B. Alexe, W.-C. Tan, and Y. Velegrakis. STBenchmark: towards a benchmark for mapping systems. *Proceedings of the VLDB Endowment*, 1(1):230–244, Aug. 2008.

[2] P. Atzeni, G. Gianforme, and P. Cappellari. A universal metamodel and its dictionary. In *Transactions on*

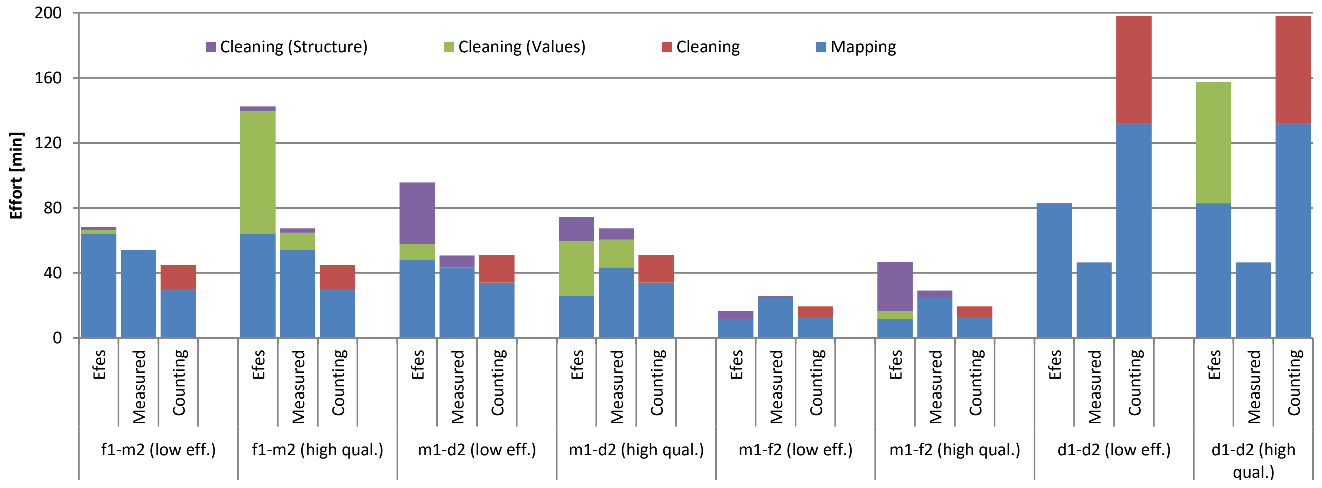


Figure 7: Effort estimates (Efes), actual effort (Measured), and baseline estimates (Counting) of the Music scenario.

Large-Scale Data-and Knowledge-Centered Systems I, pages 38–62. Springer, 2009.

[3] B. Boehm. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[4] B. Boehm, C. Abts, A. W. Brown, S. Chulani, B. K. Clark, E. Horowitz, R. Madachy, D. J. Reifer, and B. Steece. *Software Cost Estimation with COCOMO II*. Prentice-Hall, Englewood Cliffs, NJ, 2000.

[5] A. Calì, D. Calvanese, G. De Giacomo, and M. Lenzerini. Data integration under integrity constraints. *Information Systems*, 29(2):147–163, 2004.

[6] A. Calì, G. Gottlob, and T. Lukasiewicz. A general datalog-based framework for tractable query answering over ontologies. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 77–86, 2009.

[7] M. P. Consens, I. F. Cruz, and A. O. Mendelzon. Visualizing queries and querying visualizations. *SIGMOD Record*, 21(1):39–46, 1992.

[8] M. Dallachiesa, A. Ebaid, A. Eldawy, A. Elmagarmid, I. Ilyas, M. Ouzzani, and N. Tang. Towards a commodity data cleaning system. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 541–552, 2013.

[9] X. L. Dong, B. Saha, and D. Srivastava. Less is more: Selecting sources wisely for integration. *Proceedings of the VLDB Endowment*, 6(2):37–48, 2012.

[10] J. Euzenat and P. Shvaiko. *Ontology matching*. Springer-Verlag, Heidelberg (DE), 2nd edition, 2013.

[11] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: Semantics and query answering. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 207–224, Siena, Italy, 2003.

[12] R. Fagin, P. G. Kolaitis, L. Popa, and W. C. Tan:. Composing schema mappings: Second-order dependencies to the rescue. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 83–94, Paris, France, 2004.

[13] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. Mapping and Cleaning. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 232–243, 2014.

[14] B. Harden. Estimating extract, transform, and load (ETL) projects. Technical report, Project Management Institute, 2010.

[15] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: interactive visual specification of data transformation scripts. In *CHI*, pages 3363–3372, 2011.

[16] A. Kumar P, S. Narayanan, and V. M. Siddaiah. COTS integrations: Effort estimation best practices. In *Computer Software and Applications Conference Workshops*, 2010.

[17] D. MacKay. *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, 2003.

[18] B. Marnette, G. Mecca, P. Papotti, S. Raunich, and D. Santoro. ++Spicy: an opensource tool for second-generation schema mapping and data exchange. *Proceedings of the VLDB Endowment*, 4(12):1438–1441, 2011.

[19] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity flooding: A versatile graph matching algorithm. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 117–128, 2002.

[20] F. Naumann. Data profiling revisited. *SIGMOD Record*, 42(4):40–49, 2013.

[21] P. Papotti and R. Torlone. Schema exchange: Generic mappings for transforming data and metadata. *Data & Knowledge Engineering (DKE)*, 68(7):665–682, 2009.

[22] E. Rahm and H. H. Do. Data cleaning: Problems and current approaches. *IEEE Data Engineering Bulletin*, 23(4):3–13, 2000.

[23] T. Rekatsinas, X. L. Dong, L. Getoor, and D. Srivastava. Finding quality in quantity: The challenge of discovering valuable sources for integration. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2015.

[24] K. P. Smith, M. Morse, P. Mork, M. H. Li, A. Rosenthal, M. D. Allen, and L. Seligman. The role of schema matching in large enterprises. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2009.

[25] J. Wang, T. Kraska, M. J. Franklin, and J. Feng. CrowdER: Crowdsourcing entity resolution. *Proceedings of the VLDB Endowment*, 5(11):1483–1494, 2012.

Mining Frequent Co-occurrence Patterns across Multiple Data Streams

Ziqiang Yu
School of Computer Science & Technology
Shandong University
Jinan, China
zqy800@gmail.com

Xiaohui Yu[*]
School of Computer Science & Technology
Shandong University
Jinan, China
xyu@sdu.edu.cn

Yang Liu
School of Computer Science & Technology
Shandong University
Jinan, China
yliu@sdu.edu.cn

Wenzhu Li
School of Computer Science & Technology
Shandong University
Jinan, China
sduliwenzhu@gmail.com

Jian Pei
School of Computing Science
Simon Fraser University
Burnaby, BC, Canada, V5A 1S6
jpei@cs.sfu.ca

ABSTRACT

This paper studies the problem of mining frequent co-occurrence patterns across multiple data streams, which has not been addressed by existing works. Co-occurrence pattern in this context refers to the case that the same group of objects appear consecutively in multiple streams over a short time span, signaling tight correlations between these objects. The need for mining such patterns in real-time arises in a variety of applications ranging from crime prevention to location-based services to event discovery in social media.

Since the data streams are usually fast, continuous, and unbounded, existing methods on mining frequent patterns requiring more than one pass over the data cannot be directly applied. Therefore, we propose DIMine and CooMine, two algorithms to discover frequent co-occurrence patterns across multiple data streams. DIMine is an Apriori-style algorithm based on an inverted index, while CooMine uses an in-memory data structure called the Seg-tree to compactly index the data that are already seen but have not expired yet. CooMine employs a one-pass algorithm that uses the filter-and-refine strategy to obtain the co-occurrence patterns from the Seg-tree as updates to the streams arrive. Extensive experiments on two real datasets demonstrate the superiority of the proposed approaches over a baseline method, and show their respective applicability in different senarios.

1. INTRODUCTION

We study the problem of mining frequency co-occurrence patterns across multiple data streams. Given a set of unbounded streams of objects s_i ($i = 1, 2, \ldots, n$), we call a set of objects $\mathcal{O} = (o_1, o_2, \ldots, o_k)$ a frequent co-occurrence pattern (FCP) if (1) they appear in at least θ streams within a period of time no longer than τ, and (2) their appearance within each of those streams happens within a time window of size no greater than ξ, where θ, τ, and ξ are user-specified thresholds.

[*]Corresponding author.

1.1 Motivation

FCPs usually indicate strong correlations between these objects, and their timely discovery has applications in a wide spectrum of contexts. The following are some typical examples.

- *Crime prevention.* An increasing number of cities are now having traffic surveillance cameras installed on major roads and intersections for traffic management and public safety. A picture is taken when a vehicle passes by, and a structured vehicle passing record (VPRs) is sent to the data center for processing. Thus, each camera effectively produces a continuous stream of VPRs. Finding FCPs across these streams can help uncover groups of vehicles that travel together within a short time span, which is often a good indicator for potential gang crimes.

- *Discovering emerging topics.* Each user of a microblogging platform (e.g., Twitter) can be considered to produce a stream of words by posting microblogs. The intensive co-occurrence of a set of keywords in many streams (microblogs) over a short period of time can often imply the emergence of a new topic.

- *Location-based services.* Check-in apps (e.g., Foursquare) allow mobile users to check-in to the places they are located. Finding FCPs across the streams in real-time, where each stream consists of the checkin locations of a user, would help discover groups of people that are currently hanging out together, so that location-based advertising can be better targeted (e.g., offering group buying deals).

- *E-Commerce.* The browsing trace of a particular user on a e-commerce website contains a stream of items she has visited. When a set of items appear in the traces of a lot of users over a short period, it could be a sign of strong correlation

10.5441/002/edbt.2015.08

between those items, and sales strategies can be adjusted accordingly in real-time (e.g., offering combo deals for those products).

In all of the examples above, a common theme is the need to discover the frequent co-occurrence of a set of items (vehicles, people, keywords, etc.) over a short time span in multiple streams.

1.2 Challenges

Mining FCPs in realtime present some unique challenges. First, the streams are unbounded and often contain a vast volume of data with high arrival rates, allowing only one pass over the data. For example, the peak of 143,199 tweets per second were recorded by Twitter on August 3, 2013, and on average, 58 million tweets are produced per day. As another example, in the City of Jinan, a provincial capital in eastern China with a population of 6 million, the traffic surveillance cameras (each generating a stream) produce 20 million VPRs per day on average. Second, mining FCPs is a cross-stream operation, making it highly complex when many streams are involved. For instance, there are around 3,000 traffic surveillance cameras installed in Jinan; not to mention the more than 600 million active registered users of Twitter.

Existing methods for mining frequent patterns cannot be directly applied to solve the FCP mining problem. Most of the algorithms for mining frequent patterns from databases require multiple passes over the data, rendering them inapplicable in the streaming data context. Methods proposed for data streams [4, 10, 18, 15], on the other hand, focus on mining patterns from a *single* stream where the frequent patterns are defined as those that occur more often than a given threshold in *that* stream. Contrastingly, the problem we tackle is concerned with the occurrences of patterns within *multiple* streams, where the number of streams in which a pattern appears is an important parameter.

Probably the most related work to ours is that by Guo et al. [8] which considers frequent patterns in multiple data streams. But according to the their definition, whether a pattern is considered frequent totally depends on the number of its occurrences within a single stream. The frequent patterns are discovered separately from each stream, and then analysis of those patterns is performed to find the interesting ones based on their presence across multiple streams. In contrast, in order for a pattern to be considered frequent in our problem, it has to appear in at least θ streams within a short period of time. It is not the number of times a patten appearing in any single stream that matters; it is the number of streams the pattern appears in. Moreover, the algorithms proposed by Guo et al. are approximate, whereas we focus on computing exact solutions.

One seemingly promising strategy to simplify this problem is to first combine all streams into one, and then mine the FCPs on the combined data stream with some time window constraints. However, the complexity of the problem remains the same as discovering FCPs from the combined data stream still requires the differentiation between the component streams (e.g., we count a pattern twice if it appears in two different component streams, but only once if it appears twice in the same component stream), as well as restriction on the time interval of the pattern occurring in these data streams.

1.3 Our proposal

To address the above challenges, we propose two algorithms for discovering FCPs across multiple streams. They both adopt a filter-and-refine strategy with two stages. The first stage produces a set of candidate co-occurrence patterns (CPs) from the data streams, and the second stage generates the FCPs from those candidate CPs. To limit the scope of search, we divide each data stream into overlapping segments, guaranteeing that any FCP can only appear in those segments. This allows us to solve the problem of finding FCPs by focusing only on the recent segments in each stream.

As we are dealing with unbounded streams, the search for new candidate CPs is an ongoing process. The key is to efficiently search for new candidate CPs when there are newly arrived objects in any of the streams. In the DIMine approach, we introduce an inverted index called DI-Index to index the existing segments, and it computes FCPs based on the DI-Index with an Apriori-style heuristic. Although straightforward to implement, it needs to be improved in terms of memory consumption and maintenance cost.

In the CooMine approach, we propose a compact in-memory data structure called Seg-Tree to maintain the existing segments, which are dynamically updated as streams proceed, including the deletion of segments when they are guaranteed not to contain any FCPs. When a new segment is generated resulting from newly arrived objects, we search the existing segments using the Seg-Tree to find the common CPs they all contain, which form the set of candidate CPs that must be examined further. With all candidate CPs obtained, an Apriori-style algorithm is then used to compute the FCPs.

1.4 Contributions and outline

The contributions of this paper can be summarized as follows.

- For the first time, we tackle the problem of mining frequent co-occurrence patterns across data streams, an operation that has extensive applications in a variety of contexts but cannot be readily solved using existing frequent pattern mining techniques.

- We propose the DIMine and CooMine approaches with several facilities specifically designed for discovering FCPs across unbounded streams. The CooMine method includes the segmentation of data streams to limit the scope of processing, the Seg-Tree structure that compactly indexes and stores the segments for further processing, the SLCP algorithm that searches the candidate CPs for each new incoming segment, and an Apriori-style algorithm for generating FCPs from candidate CPs.

- Extensive experiments are conducted on two real data sets (a traffic surveillance data set and a Twitter data set), which demonstrate the superiority of two proposed approaches over the baseline method. Indeed, the CooMine method has been deployed in the City of Jinan's Traffic Surveillance and Public Safety Control System, and it has helped the early detection of dozens of criminal activities including vehicle thefts and burglaries over a period of six months.

The remainder of the paper is organized as follows. Section 2 reviews the related work and discusses the distinctions between our proposal and existing methods. Section 3 introduces the preliminaries, and presents the DIMine method as a first attempt to solve the FCP mining problem. Section 4 presents the Seg-tree, which is a novel index structure to maintain the valid segments. The CooMine algorithm based on Seg-tree is described in detail in Section 5. Section 6 presents the experimental results, and Section 7 concludes this paper.

2. RELATED WORK

2.1 Mining frequent patterns from databases

Mining frequent patterns (FPs) from databases is one the classic topic in data mining and has been well studied [2]. Agrawal et al. [3] present the Apriori algorithm to discover the association rules in databases, followed by a great volume of studies (e.g., [19, 18]) that also adopt Apriori-like approaches. All of them require repeatedly scanning the databases to generate candidate frequent patterns. Han et al [9] propose the FP-Tree that can discover frequent patterns without candidate generation, but the method still requires two database scans. Since the data streams are unbounded, in general only one-pass over the data is allowed, rendering those methods inapplicable.

The methods proposed by Tanbeer et al. [20] are claimed to require only a single pass over the database. However, the premise is that all relevant information is captured and stored during this pass over the database, which is then processed in later stages. This is clearly infeasible for unbounded streams.

2.2 Mining frequent patterns from data streams

Mining frequent patterns in data streams has also received considerable attention. The works based on the landmark model (e.g., [15, 21]) aim at mining frequent patterns from the start of the stream to the current moment. Other studies [4, 13, 12, 17] utilize the sliding window technique to discover the recent frequent patterns from data streams. All of these approaches obtain only approximate results with an error bound.

Some methods are proposed to obtain the exact set of recent frequent patterns from data streams [4, 12]. Chang et al [4] propose a data mining method for finding recent frequent itemsets adaptively over an online data stream with diminishing effect for old transactions. Leung and Khan propose a novel tree structure (DSTree) to capture important data from the streams to mine exact frequent patterns [12].

Although these approaches can mine frequent patterns in data streams, they cannot be directly employed to solve the FCP mining problem. Most of them focus on mining frequent patterns in a single data stream, while our task is finding FCPs across multiple data streams.

The only exception to our knowledge is the H-Stream algorithm to discover frequent patterns from multiple data streams [8]. At a first glance, that problem is very similar to ours, but indeed they are quite different. As discussed in Section 1, that work aims to search the frequent patterns that take place in multiple data streams, but does not care about the time interval of the frequent patterns occurring in these data streams. In our problem, the time interval of any FCP across multiple data streams cannot be greater than the specified threshold. Second, this work provides an approximate method but our approach can obtain the exact results. Therefore, the H-Stream algorithm cannot be applied to solve our problem.

The aforementioned existing works about mining frequent patterns from data streams always assume the streams are composed of transactions, and they can directly discover frequent patterns within transactions. But in our problem, the streams just consist of continuous unbounded objects and we have to first determine which objects in one data stream probably can construct a FCP, making the problem more complex.

2.3 Mining spatio-temporal patterns

An extensive body of literature exists on mining interesting patterns from spatio-temporal data [7, 11, 14, 5]. Some [11, 5] study the discovery of valuable trajectories of moving objects, while Li et al. [14] focus on mining spatial association rules. Moreover, there also exist works [7, 16] that study the problem of mining temporally annotated sequences. However, none of them addresses the same problem as mining FCPs, and methods proposed therein cannot be directly applied to our problem.

3. PRELIMINARIES AND A FIRST ATTEMPT

In this section, we first introduce the terminology used in the following discussion and formally define the problem, and then present a first attempt to the FCP mining problem.

3.1 Preliminaries and problem definition

DEFINITION 1. *(**Data stream**) A data stream is a continuous ordered (by timestamps) sequence of objects.*

Data streams are unbounded, and thus it is infeasible to store the data stream locally in its entirety. For an object o_i in the data stream, its ID and timestamp are id_i and t_i respectively.

DEFINITION 2. *(**Co-occurrence pattern, or CP**) For a set of objects $\mathcal{O}=\{o_1, o_2, \cdots, o_k\}$ that appear in a data stream s_i, we say that $\mathcal{O}$ is a co-occurrence pattern CP_k (where k is the number of objects) if $t_{max}^{\mathcal{O}^{s_i}} - t_{min}^{\mathcal{O}^{s_i}} \leq \xi$, where $t_{max}^{\mathcal{O}^{s_i}} = \max\{t_1, \cdots, t_k\}$, $t_{min}^{\mathcal{O}^{s_i}} = \min\{t_1, \cdots, t_k\}$, and ξ is a user-specified threshold.*

DEFINITION 3. **(Frequent co-occurrence pattern, or FCP**) *A co-occurrence pattern CP_k that appears in a set of l streams $\{s_1, s_2, \ldots, s_l\}$ is deemed a frequent co-occurrence pattern, FCP_k, if it satisfies the following conditions: (1) $l \geq \theta$, where θ is a user specified threshold; and (2) $T_{\mathcal{O}}^{\max} - T_{\mathcal{O}}^{\min} \leq \tau$, where $T_{\mathcal{O}}^{\max} = \max\{t_{max}^{\mathcal{O}^{s_1}}, \cdots, t_{max}^{\mathcal{O}^{s_l}}\}$, $T_{\mathcal{O}}^{\min} = \min\{t_{min}^{\mathcal{O}^{s_1}}, \cdots, t_{min}^{\mathcal{O}^{s_l}}\}$, and τ is a user-specified threshold and $\tau \gg \xi$.*

To understand the differences between our problem and the frequent pattern mining problem defined in earlier literature, we also present below the definition of frequent pattern in data streams given in [6].

DEFINITION 4. *(**Frequent patterns, or GHP-FP [6]**) Let the* frequency *of an itemset I over a time period (τ) in the data stream s_i be the number of transactions (where transactions correspond to non-overlapping time windows) in which I occurs. The support of I is the frequency divided by the total number of transactions observed within the time interval t_w of s_i. The itemset I is deemed a frequent pattern if its support is no less than a user-specified threshold, δ.*

We use some examples to illustrate the differences between the above definitions. In Section 1, we have discussed the scenario of each surveillance camera producing a continuous stream of VPRs, and the stream can be divided into time windows, where each window can considered as corresponding to a transaction defined in Definition 4. Consider the following cases.

Case 1: A group of cars are captured by one camera within time span ξ. This group forms a CP even if it is not captured by any other cameras.

Case 2: Within the time interval τ, a group of cars are captured by the same camera together in k non-overlapping time windows and the total number of such windows within τ is f. If $k/f \geq \delta$, this group constitutes a GHP-FP. It is not necessarily a CP unless this group of cars appear within time span ξ ($\xi \ll \tau$) at least once.

Case 3: Within the time interval τ , a group of cars are captured by m different cameras and they pass each camera within the time span ξ. If $m \geq \theta$ (where θ is a threshold that corresponds to the support δ in Definition 4), then this groups is deemed a FCP. By definition, it is also a CP.

Problem Statement. Given a set of data streams of objects and the values of parameters θ, ξ and τ, the problem of mining frequent co-occurrence patterns is to identify, on the fly, the FCPs within the streams as streams evolve over time.

To facilitate the search for FCPs, we divide each stream into overlapping *segments*. Each segment is a sequence of objects ordered by their timestamps, and the time span of a segment is the time interval between its first and last objects.

DEFINITION 5. (**Segment**) *For a given data stream s, a segment $G(o_1 o_2 \cdots o_m)$ is a subsequence of s that satisfies both of the following conditions:*

(1) $|t_i - t_j| \leq \xi$, $\forall$ o_i, o_j in G; and

(2) The time span of G must be maximal with respect to ξ. That is, that does not exist a subsequence of s, G', such that G' is a segment and G is a strict subsequence of G'.

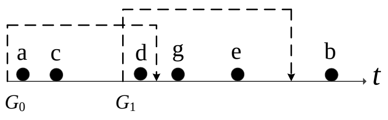


Figure 1: A segment. The temporal relationship between the objects is as follows: $t_d - t_a < \xi$, $t_g - t_a > \xi$, $t_g - t_d < \xi$, $t_g - t_c > \xi$, $t_e - t_d < \xi$, $t_b - t_d > \xi$.

Example 1. Figure 1 shows a stream of objects, where *a* is the first object of the segment G_0. Because $t_d - t_a < \xi$ and $t_g - t_a > \xi$, *d* is the last object of G_0. Note that the objects *c*, *d*, and *g* do not constitute a segment because $t_g - t_c > \xi$.

DEFINITION 6. (**Prefix of the segment**) *For a given segment G_i $(o_1 o_2 \cdots o_m)$, its prefix is a subsequence of G, $o_1 o_2 \cdots o_j$, where $1 \leq j \leq m$.*

The definition of segment dictates that any data stream can be uniquely partitioned. In addition, any CP must be covered by some segment(s). Therefore, the problem of mining FCPs accross multiple streams can be converted to finding the FCPs contained in the segments of the data streams.

3.2 A first attempt: the DIMine approach

Our first attemp to solve the problem of mining FCP is a simple method called DIMine. Following the discussion in Section 3.1, the basic approach is to divide the data streams into segments as the streams evolve, and then discover FCPs from those segments. To this end, we use an inverted index called DI-index to index all existing segments, and also maintain a list storing the information related to each segment including its starting and ending times as well as the ID of the data stream it belongs to. The DI-index can be implemented as a hash map, with each entry taking the form of $(o_i, \mathcal{G}_i)$, where o_i is an object and $\mathcal{G}_i$ stores the set of IDs of the segments containing o_i.

For an incoming segment G, DIMine finds the newly formed FCPs caused by G in the following steps using the anti-monotone Apriori heuristic based on the DI-Index.

(1) For each object o_i in G, it finds the entry $(o_i, \mathcal{G}_i)$ from the DI-Index and then sets $\mathcal{G}_i = \mathcal{G}_i \cup G$, It then verifies whether o_i is a FCP$_1$ (a FCP with only one object) by straightforward counting based on $\mathcal{G}_i$.

(2) It next iteratively generates the set of candidate FCPs with (k+1) objects from FCP$_k$ ($k \geq 1$) caused by G, and picks those that conform to the FCP definition. Assuming that P_{k+1} is a candidate FCP with the set of objects $\{o_1, \cdots, o_{k+1}\}$ and $\mathcal{G} = \mathcal{G}_1 \cap \cdots \cap \mathcal{G}_{k+1}$, if the segments in the set $\mathcal{G}$ involve no less than θ data streams within the time interval τ , then P_{k+1} is a FCP.

As the DI-Index is a hash map, the DIMine approach can find the segments containing a given object o with constant time. Also, the use of the Apriori heuristic helps prune the search space, making the algorithm more efficient.

In addition to finding the FCPs, we also need to worry about the maitenance of the DI-Index, as streams constantly evolve over time, and some objects may become obsolete. The DIMine approach needs to scan all entries of the DI-index frequently to remove the identifiers of the obsolete segments because the expired data will not only cause false positive results but also increase the memory consumption. This task incurs non-trivial cost. Assuming that in a stable state, the number of segments being indexed is n and the average number of objects in a segment is p, then it takes $O(n \cdot p)$ time to detect all expired segments. As will be shown in the experimental results in Section 6, this is actually much more expensive than finding the FCPs.

4. THE SEG-TREE

Addressing the problems with DI-Mine, we present the CooMine approach that has better maintenance efficiency with an index that is more compact when significant overlapping exists between adjecent segments. We call this index the Seg-Tree.

4.1 Motivation

There are two critical issues to be addressed in mining FCPs across multiple data streams. First, as a large volume of segments may be generated at a varying rate, it is important to effectively index them with in a space-efficient manner for further processing. Second, for a new incoming segment, not every existing segment can form FCPs with it; thus it is necessary to quickly narrow down the search space. As such, we need an effective index structure that meets the following requirements: (1) good support for mining FCPs; (2) efficient handling of the insertion of new segments and deletion of obsolete segments; and (3) being frugal with its memory usage.

4.2 Overview of the Seg-tree structure

To satisfy the above requirements, we propose the Seg-Tree, a main-memory-based structure, to manage the valid segments. A segment G is valid, if and only if $t_{now} - t_G^f \leq \xi$, where t_{now} is the current time and t_G^f is the timestamp of the first object of G.

The Seg-tree index structure has three components: the *Seg-tree* itself and two auxiliary structures,*Hlist* and *Tlist*. Figure 2 shows a Seg-tree that indexes the segments in Figure 3.

The Seg-tree is a trie-like structure (but with notable differences) with its nodes corresponding to the objects. (We hereinafter use nodes and objects interchangeably when there is no ambiguity.) Edges exist between objects that appear adjacent to each other in some segment. Segments can be continuously inserted based on their prefixes, and obsolete segments can be removed as time progresses. The nodes are doubly-linked between child and parent. The details of insertion and deletion will be discussed later in Sections 4.4 and 4.5.

The two auxiliary structures are used to quickly locate specific nodes in the Seg-tree. We maintain a separate linked list of references to the nodes corresponding to the same object, and the head of each list is stored in Hlist. In Figure 2, we show the Hlist containing the heads for some of the lists.

Tlist, on the other hand, stores references to all the tail nodes of the Seg-tree, where a tail node refers to the last object of a segment. Each node in the Tlist is a reference to a tail node in the Seg-tree.

Once a tail node expires, we can use Tlist to quickly locate the corresponding segment in the Seg-tree and remove it. Figure 2 shows the links of tail nodes k and o.

4.3 Node structure

For a segment G_j, its last object is represented in the Seg-tree by a tail node tn_j and other objects by *ordinary nodes*. An ordinary node has four attributes $\langle Id$, *object*, *distance*, *count*, *reference*$\rangle$), where Id is the identifier of the node, *object* is the reference to the object represented by this node, *distance* represents the distance (number of edges) from this node to tn_j, *count* records the number of segments containing this node, and $reference$ points to the next node corresponding to the same object in the Seg-tree. For the new segment G_j, for each node, $distance$ can be easily calculated, *object* is known, $count$ is set to 1, and *reference* is initialized to null.

The tail node tn_j has five attributes $\langle Id$, *object*, *distance*, *count*, *reference*, $\mathcal{L}\rangle$. Here, *Id*, *distance*, *count*, and *reference* have the same meaning as that for ordinary nodes. $\mathcal{L}$ is a set of tuples $\{G_j, l_j\}$, where G_j is the segment with tn_j being the tail node and l_j is the length of segment G_j. Because tn_j may be the tail nodes of multiple segments, for each segment G_j, we have to record its length l_j.

With the help of Tlist, the Seg-tree structure has the following property. For the segment G_i, since tn_i records the length (l_i) of G_i, backtracking l_i-1 steps from tn_i to the root of the Seg-tree can uniquely determine the segment G_i, which is linear with respect to the length of the segment.

4.4 Constructing the Seg-tree

The Seg-tree is initialized as a root node (*null*). A new segment G_j can be inserted into the Seg-Tree using the following steps.

- We search for the longest matching prefix (pre_j) of G_j in the Seg-tree, the details of which will be discussed later.
- If pre_j exists, the remaining objects in G_j are appended to pre_j. For a segment, its longest matching prefix probably exists in different branches of the Seg-tree. In case this happens, the first being found will be picked. Otherwise, G_j will be directly added to the root.
- If pre_j exists, we need to update attribute values of the nodes in pre_j after G_j being inserted into the Seg-tree; the attribute values of other nodes remain unchanged.
- We insert the tail node tn_j into Tlist, and append each node of G_j to the respective linked list for the corresponding object.

Example 2. In Figure 2, the Seg-tree manages the segments in data streams s_1 and s_2 (as shown in Figure 3). Now we describe the process of inserting the segments from s_1. At the beginning, the Seg-Tree contains the root only. When the segment G_0 (*b, c, d*) appears, it is added to the root. As to the segment G_1 (*c, d, f, k*), since its prefix (*c, d*) exists in the tree, the objects *f, k* can be appended to the existing prefix. The segment G_2 (*h*, *m*, *n*) has no matching prefix, so it is inserted at the root. The segments G_3 (n, c, p, o) and G_4 (h, b, k, r, s, t) also have existing matching prefixes n and h separately in the Seg-tree, so they can be appended to their existing prefixes.

The Seg-tree is similar to *trie* [1], but they have significant differences. In a trie, the children of any node must have the common prefix. But in the Seg-tree, if a segment has a matching prefix in any branch of the Seg-tree (not necessarily starting from the root), then this segment can be appended to the existing prefix that are shared by multiple segments. Because there may be extensive overlapping between adjacent segments, this sharing can be quite effective in saving memory. For the trie, on the other hand, most of the overlapping segments cannot be compacted because they start with different objects.

Algorithm 1 *Searching prefix* Algorithm

```
Require:
    The new segment $G_j$, the Seg-Tree;
Ensure: :
    The Seg-Tree after $G_j$ being inserted;
 1: $\mathcal{H}$=Hlist($o_1$); //$\mathcal{H}$ is the set of nodes that have the same identi-
    fier with the first node $o_1$ of $G_j$
 2: $pre_j$=$\emptyset$;
 3: for $n_i$:$\mathcal{H}$ do
 4:     $pre_i$=$pre_i$+$n_i$; count=1; $n_j$=$n_i$;
 5:     while $count < G_j.length$ do
 6:         flag=false; count++;
 7:         $\mathcal{N}_c$= child nodes of $n_j$;
 8:         for $n_c$:$\mathcal{N}_c$ do
 9:             if $n_c$==$G_j$.next() then
10:                 $pre_i$=$pre_i$+$n_c$; $n_j$=$n_c$; flag=true;
11:                 break;
12:             end if
13:         end for
14:         if flag==false then
15:             break;
16:         end if
17:     end while
18:     if $pre_j.size < pre_i.size$ then
19:         $pre_j$ =$pre_i$;
20:     end if
21: end for
```

In the aforementioned procedure of building the Seg-tree, two issues need to be discussed further.

(1) *Searching matching prefixes.* Searching the matching prefixes in the Seg-tree for an incoming segment is critical for its insertion. We can utilize the Hlist to accelerate this process. To search the matching prefix (pre_j) of a segment G_j ($o_1 o_2 \cdots o_{l_j}$), we first determine the set ($\mathcal{H}$) of nodes matching o_1 based on the Hlist. Next, for each node n_r ($n_r \in \mathcal{H}$), we shall traverse its children to find the node matching o_2. If the child n_c matches o_2, then $\{n_r, n_c\}$ is the current matching prefix of G_j, and we only need to scan the children of n_c to expand pre_j, while other children of n_r can be safely discarded because it is impossible for n_r to have two children that correspond to the same object. In this iterative fashion, pre_j can be determined. The details of searching prefixes are shown in Algorithm 1.

THEOREM 1. *For a given segment G_j with length l_j, the time complexity of searching for the longest matching prefix of G_j from the Seg-tree is $O(l_j)$.*

PROOF. Suppose that there are f nodes corresponding to the object o_1 (the first object of G_j). For any node n_r matching o_1, we at most traverse l_k levels of the subtree rooted at n_r. Hence, the dominating time cost of searching the longest existing prefix of G_j is $f \cdot l_k \cdot t_p$, where t_p is the unit time of traversing one level of the subtree. Since the parameters f and t_p are constants, the time complexity of searching the prefix of G_j is $O(l_j)$. □

(2) *Updating the attribute values of the nodes being inserted.* When G_j is inserted into the Seg-tree, for any node $n_i \in pre_j$, its attribute values have to be updated. Assuming that ($distance_i$, $count_i$ and $reference_i$) and ($distance'_i$, $count'_i$ and $reference'_i$)

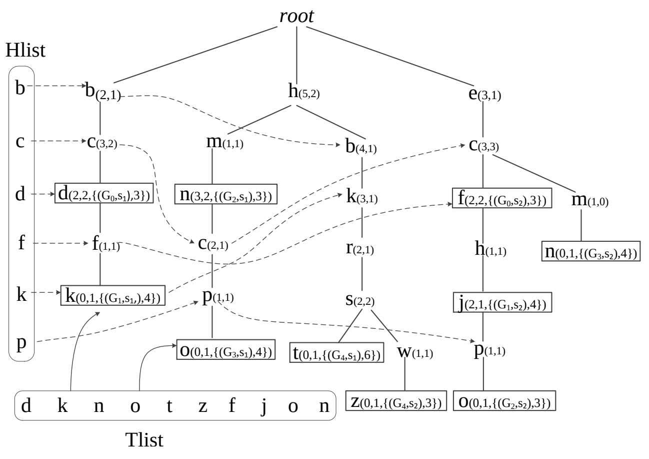


Figure 2: A sample Seg-tree. In this tree, the node in bold boxes are tail nodes and others are ordinary. The node annotation $h_{(5,2)}$ indicates that the distance between h and the farthest tail node t of G_4 is 5, and it appears in two segments. For the tail node $t_{(0,1,((G_4,s_1),6))}$, the tuple $((G_4, s_1)$,6)) represents that t is the tail node of the segment G_4 in s_1, and the length of G_4 is 6.

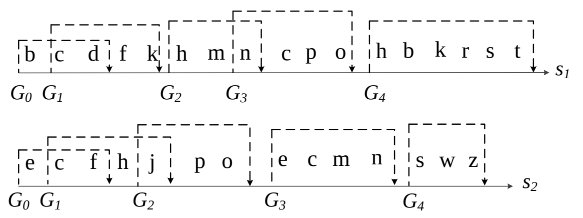

b c d f k h m n c p o h b k r s t
s_1
G_0 G_1 G_2 G_3 G_4
e c f h j p o e c m n s w z
s_2
G_0 G_1 G_2 G_3 G_4

Figure 3: The segments managed by the Seg-tree shown in Figure 2

are the original and updated attribute values of n_i, the attribute values can be updated as follows.

- $distance'_i$=$max\{distance_i, distance'_i\}$. If n_i belongs to multiple segments, $distance'_i$ records the distance between n_i and the farthest tail node of the segments containing n_i.
- $count'_i$=$count'_i + 1$, which means that the number of segments that n_i appears in increases by one.
- $reference'_i$ is still null. If a node n'_i corresponding to the same object as n_i is inserted later, then $reference'_i$ will point to n'_i.
- If the tail node tn_j also belongs to pre_j, then aside from doing the above updates, we also need to add the tuple $\{G_j, l_j\}$ into $\mathcal{L}'_j$, that is, $\mathcal{L}'_j = \mathcal{L}'_j \cup \{G_j, l_j\}$.

Example 3 In Figure 2, before inserting the segment G_1 from stream s_1, the $distance$ and $count$ values of c and d in the Seg-tree are $(1, 1)$ and $(0, 1)$ separately. When G_1 is inserted, since c and d belong to the matching prefix of G_1, the values will be updated to (3,2) and (2,2) respectively.

4.5 Removing obsolete segments from the Seg-tree

As streams proceed, some segments in the Seg-tree will become obsolete. These obsolete segments will waste memory and have negative influence on mining FCPs, and they thus should be removed from the Seg-Tree in time, taking into the cost of removal. On one hand, the existing segments become obsolete frequently, so it is unwise to continuously monitor all segments and delete a segment as soon as it becomes obsolete. On the other hand, if the expired segments are retained in memory for too long, its impact on the memory consumption as well the search efficiency will become an issue.

As a good trade-off between effects of obsolete segments and the deletion cost, a Lazy Deletion (LD) strategy is introduced. In this strategy, we do not delete all expired segments at once, but only remove those that are relevant to the new incoming segment that needs to be processed. The deletion can also be triggered by the used memory exceeding the specified threshold, at which time we will scan the Tlist to find and remove all obsolete segments. Here, the relevant obsolete segments for a new segment can be determined by Algorithm 3 to be presented in Section 5.

Deleting an obsolete segment G_e involves two specific tasks: (1) decreasing the attribute value *count* of nodes in G_e and removing the nodes with *count* being zero from the Seg-tree; and (2) inserting the disconnected subtrees into the Seg-tree. Deleting G_e can possibly cause one or more subtrees being disconnected from the Seg-Tree, and these subtrees need to be added back into the Seg-Tree. For a disconnected subtree, we define its *single prefix* as the path from the root to the first node with more than one child. We treat the single prefix as a segment and insert it into the Seg-tree using the insertion method. The other objects of the disconnected subtree are appended to the prefix. In this way, any disconnected subtree can be inserted back into the Seg-tree.

The deletion cost can be reduced with the help of the Tlist. In the Tlist, we order the tail nodes by their arrival time, and thus we only need to determine the latest obsolete tail node, and then we can infer that all the tail nodes preceding this node are all obsolete. Hence, the obsolete segments can be quickly determined.

4.6 Comparison of Seg-tree and FP-tree

We are inspired by the FP-tree [9] in designing the Seg-tree to support the mining of FCPs. Compared with FP-tree, the Seg-Tree has the following advantages for the problem addressed in this paper.

(1) To construct the FP-tree, the objects in the transactions need to be sorted according to their counts. But in the Seg-tree, the objects in the segments do not need to be sorted, saving on the sorting cost. To be more specific, the count of each object changes frequently as data streams evolve, and therefore the order of objects in existing transactions needs to be adjusted constantly as required by the FP-tree. The FP-tree thus has to be updated or rebuilt frequently, causing inhibitive maintenance cost. As such, the FP-tree is not suitable for mining FCPs across data streams in real-time.

(2) In our problem, there may exist extensive overlapping between nearby segments. If we employ the FP-tree to index them without sorting, no compaction can be achieved according to the insertion rules defined in [9]. When the Seg-tree is used, however, many overlapping segments can be compacted tightly because they have common matching prefixes in the Seg-tree.

(3) The FP-tree deals with static datasets and does not specifically consider the effect of frequent updates, while in our case, the Seg-Tree has to be constantly updated and thus efficient maintenance cost is vital. By adopting the LD strategy to delete the obsolete data at different time granularities, we strive to achieve a balance between memory consumption and deletion cost.

5. THE COOMINE APPROACH

We propose the CooMine approach that can utilize the Seg-tree for mining FCPs. We first give an overview of this approach, and then describe each component in more detail.

5.1 Overview of the CooMine approach

As discussed in Section 3, any FCP is covered by at least θ segments. We thus propose the CooMine approach to mine FCPs from the segments in each data stream. Since the streams are constantly changing with newly arrived objects, the task of mining FCPs is a continuous process, with actions triggered by the creation of each new segment as new objects arrive. For the new segment G_j, the *CooMine* approach consists of two components.

(1) *Searching for the largest common CPs*: If the new segment G_j can form new FCPs with existing segments, the FCPs must be covered by their common CPs. The largest common CP between any two segments refers to the largest set of common objects between them, so any common CP of two segments must be covered by their largest common CP. Therefore, we design an algorithm to find the largest common CPs between G_j and the existing segments to narrow down the search scope. Hereinafter, we use LCP to represent the longest common CP.

(2) *Mining FCPs from the LCPs*: Since the set of LCPs is finite and its size is usually small, mining FCPs boils down to the problem of finding the CPs that appear in at least θ data streams within time interval τ. This problem can be solved with an Apriori-style algorithm.

5.2 The SLCP algorithm for searching LCPs

A naive method to find the LCP is to compare the new segment with every existing segment. However, since only a few existing segments have common CPs with the new segment, comparisons with many segments are wasteful. Also, the cost will be huge when the Seg-tree is large.

To cut down the search cost, we design the *SLCP* algorithm to find the LCPs between the new segment G_j and existing segments. This algorithm first searches the *relevant segments* for each node of G_j, where the relevant segments are defined below in Definition 7. Next, it can deduce which common nodes with G_j each relevant segment has. The set of common nodes between each relevant segment and G_j is a LCP.

DEFINITION 7. (Relevant segment): *For any node n_i (except the root) in the Seg-tree, if the segment G_i contains n_i, then G_i is a relevant segment of n_i, and tn_i is a relevant tail node of n_i.*

Algorithm 2 describes the SLCP algorithm. First, for each node n_i of G_j, we find all nodes that have the same identifier with n_i based on Hlist (Line 4). Second, for any node n_j that corresponds to the same object as n_i, we determine its valid relevant segments (Line 5). Third, once the relevant segments of each node are determined, the LCPs are computed (Line 6-8).

In the SLCP algorithm, discovering the LCPs is a non-trivial problem. In the procedure of building the Seg-tree, any segment itself will be discarded after being inserted into the Seg-tree. Meanwhile, the ordinary nodes of the Seg-tree no longer record the information of their relevant segments for memory saving. In this case, for any ordinary node, we cannot directly determine the segments containing it. Therefore, we need to find an efficient way to determine the relevant segments for the specified nodes.

5.2.1 Searching relevant segments

We propose the *DistanceBound* method that can efficiently find relevant segments for the specified nodes. The *DistanceBound* method can prune the search scope by utilizing the distance between each node and its furthest relevant tail node as bound to accelerate searching relevant segments for the specified node. For the specified node

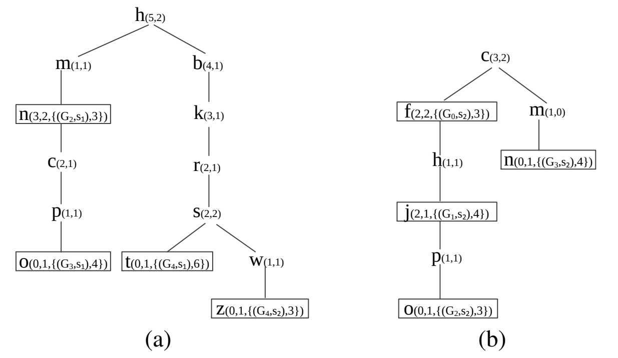


Figure 4: Two subtrees of the Seg-tree

n_j, this method visits n_j and guarantees that by visiting Dis_j levels of the tree rooted at n_j, it can find all relevant tail nodes for n_j according to Theorem 2. When this method visits any child n_c of n_j, only $min\{Dis_c, Dis_j - 1\}$ levels of the tree rooted at n_c need to be traversed. In this way, the search algorithm can quickly converge and the search space can be reduced.

Algorithm 3 describes the process of searching relevant segments for n_j, which consists of four steps:

Step 1: Build a queue *Q* and enqueue the pair $\langle n_j, stepcount\rangle$ into *Q*, where *stepcount*=Dis_j. (Lines 1-4)

Step 2: Get the first pair $\langle n_x, stepcount\rangle$ from *Q*. (Line 6)

Step 3: If n_x is a relevant tail node of n_j, insert n_x into the set $\mathcal{R}$. Otherwise, for each child n_c of n_x, set *stepcount* as $min\{Dis_c$, *stepcount*-1$\}$ and enqueue $\langle n_c, stepcount\rangle$ into *Q* if $stepcount \neq 0$. (Lines 7-15)

Step 4: If *Q* is not empty, go to step (2); otherwise, return the set $\mathcal{R}$ and the search ends. (Line 5)

In Step 3 , for the tail node n_x, if we assume that its corresponding segment is G_x with length l_x and the distance from n_j to n_x is D_{jx}, then G_x is a relevant segment of n_j if l_x is no less than D_{jx} and G_x is valid.

THEOREM 2. *For any node n_j in the Seg-tree, all relevant tail nodes of n_j can be found by visiting at most Dis_j levels of the tree rooted at n_j.*

PROOF. Suppose that there is a tail node tn_j that cannot be discovered even if after taking (Dis_j-1) steps in each branch of n_j. Since n_j and tn_j belong to the same segment, tn_j and n_j must be in the same branch. Because tn_j cannot be found by taking Dis_j steps from n_j in this branch, the distance from n_j to tn_i must be greater than Dis_j. However, Dis_j is the distance between n_i and the farthest relevant tail node. Contradiction. $\square$

The *DistanceBound* algorithm can effectively prune the search space based on the attribute $distance$ of nodes. Figure 4 shows two subtrees of the Seg-tree in Figure 2. In Figure 4(a), the node h has attribute Dis_h as 5; the *DistanceBound* algorithm thus probably needs to take 5 steps in its each branch to find the relevant tail nodes for h. However, when it visits the node m and finds that Dis_m is equal to 1, it then only needs to take one step in the branch of m to search the relevant tail node n. The nodes after n do not need to be scanned. As to the node c in Figure 4(b), since Dis_c is 3, we only need to traverse its left branch by 3 steps to search for the relevant tail node j, and nodes after j can be ignored.

5.2.2 Obtaining the LCPs

For each node n_i in G_i, all relevant segments of n_i can be determined by the *DistanceBound* algorithm. If we employ a hash table

Algorithm 2 Searching for LCPs (*SLCP*)

```
Require:
    The new segment $G_j$, the Seg-Tree;
Ensure: :
    The LCPs between $G_j$ and existing segments;
1: $\mathcal{H}_i$=null; // a set of nodes have the same identifier with $n_i$
2: The Map$< G_i, P_i >$ map=null; // $P_i$ is the LCP between $G_j$ and $G_i$
3: for $n_i$:$G_j$ do
4:     $H_i$=Hlist($n_i$);
5:     $\mathcal{R}_i$=DistanceBound ($H_i$); // $\mathcal{R}_i$ includes segments covering $n_i$
6:     for $G_i$:$\mathcal{R}_i$ do
7:         map.add($< G_i, n_i >$);
8:     end for
9: end for
10: Output map;
```

Algorithm 3 *DistanceBound* Algorithm

```
Require:
    $H_i$;
Ensure: :
    The relevant segments of $n_i$;
1: stepcount=0, $\mathcal{R}$=null, Queue=null;
2: for $n_k$:$H_i$ do
3:     stepcount=$Dis_k$ and generate the pair $\langle n_k$, stepcount$\rangle$;
4:     Queue.put($\langle n_k$, stepcount$\rangle$);
5:     while Queue$\neq$ empty do
6:         $n_f$=Queue.getfirst();
7:         if $n_f$ is a tail node and $G_f$ covers $n_k$ and $G_f$ is valid then
8:             $\mathcal{R}_k$.add($G_f$)
9:         end if
10:        for $n_c$:children of $n_f$ do
11:            stepcount=min$\{Dis_f, (stepcount - 1)\}$;
12:            if stepcount $\neq 0$ then
13:                Queue.put($\langle n_c$, stepcount$\rangle$);
14:            end if
15:        end for
16:    end while
17:    $\mathcal{R}$=$\mathcal{R} \cup \mathcal{R}_k$;
18: end for
19: return $\mathcal{R}$;
```

to store the relevant segments with the key being the ID of each relevant segment and the value being a list of nodes common to this segment and G_i, then the largest set of common nodes between each relevant segment and G_i can be immediately determined, and the largest set of common nodes is a LCP between G_i and the corresponding relevant segment.

5.3 Mining FCPs from the LCPs

5.3.1 Mining FCPs with the Apriori heuristic

For the new incoming segment G_j, if it can form new FCPs with existing segments, then these FCPs must be covered by their LCPs. Therefore, we can mine the FCPs from the these LCPs with the anti-monotone Apriori heuristic. The basic idea is to iteratively generate the set of FCPs with $(l + 1)$ objects based on the set of FCPs with l $(l \geq 1)$ objects. Theorem 3 guarantees the correctness of the CooMine algorithm.

Table 1: Summary of common CPs

LCPs	segments
$\{m, n, \}$	$\{G_2, s_1\}$
$\{n, p, o\}$	$\{G_3, s_1\}$
$\{p, o\}$	$\{G_2, s_2\}$
$\{m, n\}$	$\{G_3, s_2\}$
$\{n\}$	$\{G_4, s_2\}$

Algorithm 4 CooMine Algorithm

```
Require:
    The new segment $G_i$, The Seg-Tree;
Ensure: :
    The FCPs of length $k$ formed by $G_i$;
1: LCPtable = SLCP($G_i$, Seg-tree);
2: $l$=1;
3: while $l < k$ do
4:     if $l = 1$ then
5:         Obtaining FCPs of length $l$ based on the LCPtable;
6:     else
7:         Generating candidate FCPs with length $l$ based on the FCPs of length $l - 1$;
8:         Detecting each candidate FCP based on the LCPtable and discover the FCPs of length $l$;
9:     end if
10: end while
11: Output the FCPs of length $k$.
```

THEOREM 3. *If a set of objects $\mathcal{O}_n$ is a FCP, then any of its subset $\mathcal{O}_n^{'}$ must also be a FCP.*

Since LCPs are found from valid segments, the time span of all obtained LCPs must be no greater than the threshold τ. According to the Definitions 2 and 3, if a CP occurs in more than θ data streams within the time interval τ, then this CP must be a FCP. Hence, the CooMine algorithm only needs to consider the number of data streams that the LCPs appear in. Because all LCPs between T_k and existing segments have been found by the *SLCP* algorithm, the CooMine algorithm (Algorithm 4) only needs to mine all FCPs from the LCPs.

Example 4. Assuming that G_0 (*mnpo*) is a new segment in data stream s_3 and the parameters k and θ (cf. Definition 3) are 2 and 3 separately. Table 1 shows the LCPs between G_0 and existing segments in Figure 2. Based on Table 1, CooMine can find the FCPs of size 1 ($\{m\}$, $\{n\}$, $\{o\}$, $\{p\}$), and then deduce the FCPs of size 2 ($\{m, n\}$, $\{p, o\}$). Since only one LCP has three objects, we assert that there does not exist a FCP of size 3.

5.4 Advantages of our approach

CooMine has the following advantages.

(1) The use of the Seg-tree to index the valid segments can help save on memory consumption. The segments are inserted into the Seg-tree based on the prefix rule, and many common objects in multiple segments can be compacted to save memory. In addition, the ordinary nodes in the Seg-tree do not record the segments that they belong to and only the tail nodes maintain this information. Since ordinary nodes make up the largest part of the Seg-tree and each ordinary node always takes place in multiple segments, omitting the segments information from ordinary nodes can save much on memory consumption.

(2) The maintenance cost of the Seg-tree is low. A new segment can be inserted into the Seg-tree with a small cost because its

Table 2: The parameters used in the experiments

parameters	meaning
ξ	the time interval in Definition 2
k	the number of objects in one FCP
τ	the time interval in Definition 3
θ	the number of streams in Definition 3
D_s	the scale of data

matching prefix can be quickly determined; the obsolete segments can be immediately determined with the help of the Tlist, and the LD strategy can remove the obsolete data at different granularities to reduce the deletion cost.

(3) CooMine first searches the LCPs covering all FCPs between each new segment and existing ones; this step can effectively narrow down the search scope for mining FCPs.

6. EXPERIMENTS

We conduct experiments to evaluate the performance of DIMine and CooMine methods, and compare them with the MatrixMine algorithm that is introduced as a baseline method. The parameters involved in experiments are illustrated in Table 2.

Specifically, we first evaluate the index structures employed in three methods with respect to memory consumption and maintenance cost, and then compare the performance of the three methods for computing FCPs. Finally, we test the influence of varying parameters on the CooMine algorithm in the following aspects: the time of discovering FCPs, the sustainable workload, and the number of FCPs.

In our experiments, we implement three methods (CooMine, DIMine, and MatrixMine) using Java and the indexes of three methods all employ the standard Java Collection classes as the storage structure to record the segments information. To make the results more accurate, every evaluation is repeated ten times and the average values are recorded as the final results.

6.1 Experimental setup

The experiments are conducted on a Dell OPTIPLEX 990, a PC with a 3.1GHz Intel i5-2400 processor and 8GB RAM. Two real datasets are used. One dataset is a traffic records dataset (TR dataset) of Jinan city in China. The TR dataset contains all VPRs (vehicle passing records) of each monitoring camera in Jinan on May 1, 2013. For each monitoring camera, we simulate its passing records as a data stream, then the TR dataset can be viewed as multiple data streams. For the TR dataset, D_s represents the number of VPRs, and each vehicle (identified by its license plate) is an object.

The other dataset (the Twitter dataset) is provided by Twitter for academic research purposes[1]. In Twitter dataset, each word is considered an object and a tweet corresponds to a segment. In this case, all of the objects in a segment appear at once, as a tweet as a collection of words is posted as a whole. The tweets by the same user constitute one data stream. D_s represents the number of all segments (tweets).

6.2 The baseline method

Since none of the existing methods is applicable to the problem of finding FCPs in real-time either because they address a different problem or because their cost is apparently too expensive when applied to this problem, we introduce the MatrixMine algorithm as the baseline method, and compare it with our proposed methods w.r.t. memory consumption, maintenance cost, mining time, and total cost.

MatrixMine also divides the data stream into segments and then mines FCPs from the segments. The information of all segments are maintained in an independent list and we can obtain the information of each segment based on its identifier from this list. MatrixMine maintains a matrix M that keeps track of each pair of co-occurring objects. If we assuming that there are n distinct objects ($\{o_1, \cdots, o_n\}$) in a particular application, then M is a $n \times n$ matrix, where each element $c_{i,j}$ of M corresponds to the pair $p_{i,j}$ that consists of objects o_i and o_j ($1 \leq i, j \leq n$). Because the pair $p_{i,j}$ probably occurs in different segments, the corresponding element $c_{i,j}$ has to maintain a set $\mathcal{I}_{i,j}$ that contains multiple tuples and each tuple is represented as $\langle G_j, s_h \rangle$, where G_j is the identifier of the segment that $p_{i,j}$ belongs to, and s_h is the data stream containing G_j.

For any pair $p_{i,j}$ of the new segment G_j, MatrixMine can determine whether $p_{i,j}$ is a FCP based on $\mathcal{I}_{i,j}$ because it records the identifiers of all segments covering the pair $p_{i,j}$. Once the FCPs with two objects are obtained, MatrixMine can iteratively generate the FCPs with more objects employing the Apriori heuristic.

The maintenance of the matrix M includes the insertion of newly generated pairs and the deletion of obsolete pairs. When a new pair $p_{i,j}$ appears, it must be inserted into M right away. To reduce the deletion cost of the Matrix structure, we also adopt the Lazy Deletion strategy to remove the obsolete data from the matrix.

6.3 The performance of index structures

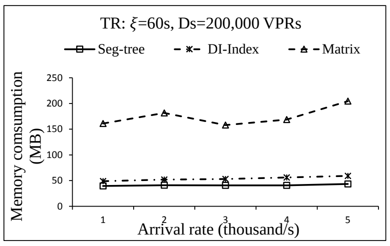
We compare the Seg-tree, the DI-Index, and the *Matrix* with respect to memory consumption and the maintenance cost, and the results are shown in Figure 5.

Memory consumption. Figure 5(a) shows the memory consumption of index structures on the TR dataset. Here, we test the memory consumption for processing the incoming data within one second with varying arrival rates when D_s is fixed, where D_s represents the volume of data that has been processed. The experimental results demonstrate that the memory consumed by Seg-tree is about 80% of that consumed by DI-Index and only 25% of that by *Matrix* because of the overlap between segments makes it possible to compact them tightly in the Seg-tree. *Matrix* consumes much more memory than Seg-tree and DI-Index, and the reason is the *Matrix* has to maintain a large volume of elements.

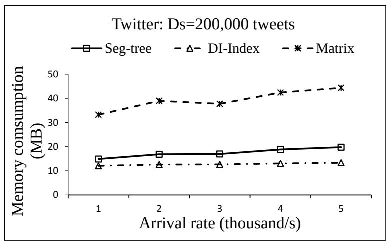
Figure 5(b) shows the result on the Twitter dataset, where the Seg-tree consumes slightly more memory than DI-Index. The reason is that most adjacent segments (tweets) do not overlap, and the Seg-tree cannot compact them as well as for TR. However, the Seg-tree still consumes much less memory than the *Matrix*.

Maintenance cost. Figure 5(c), 5(d) and 5(e) show the maintenance time of processing the data within one second with different arrival rates for the two datasets. In Figure 5(c), we evaluate the maintenance cost of the Seg-tree, DI-Index, and *Matrix* with the fixed values of the parameters ξ, τ, and D_s . The results demonstrate that the maintenance cost of the Seg-tree is much smaller than that of DI-Index and *Matrix*, while the *Matrix* has the largest maintenance cost. Specifically, the maintenance cost of the Seg-tree is approximately 50% of that consumed by DI-Index, while the maintenance cost of *Matrix* is almost 20 times as big as that of Seg-tree. In the Seg-tree, we can immediately determine the expired segments based on the Tlist and remove them, which can save much maintenance time. But in DI-Index and *Matrix*, we have to detect each element to remove the obsolete data, increasing the maintenance cost. Since the number of elements in *Matrix* is greater than that of elements in DI-Index, maintaining *Matrix* is more expensive than maintaining the DI-Index.

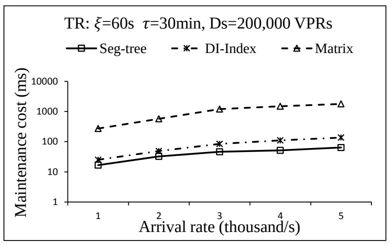
[1]Tweets2011. http://trec.nist.gov/data/tweets/



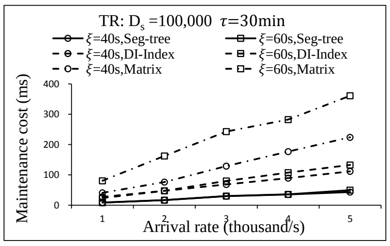
(a) Memory consumption (TR)



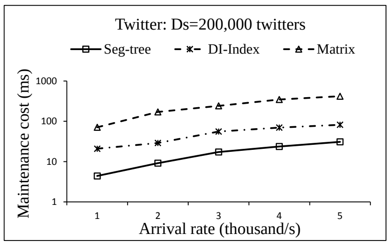
(b) Memory consumption (Twitter)



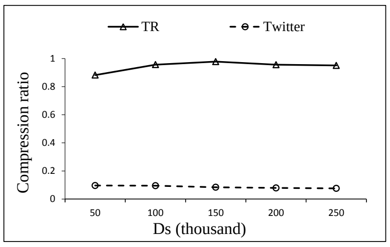
(c) Maintenance cost w.r.t D_s (TR)



(d) Maintenance cost w.r.t ξ (TR)

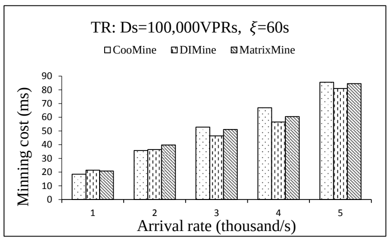


(e) Maintenance cost w.r.t D_s (Twitter)

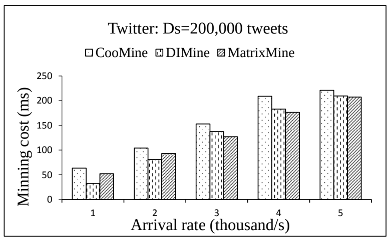


(f) Compression ratio w.r.t ξ

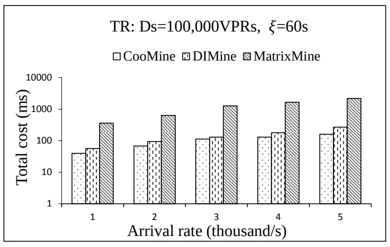
Figure 5: Evaluation of index structures



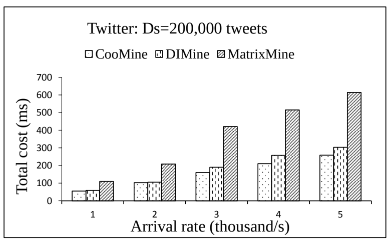
(a) Comparison of mining cost (TR)



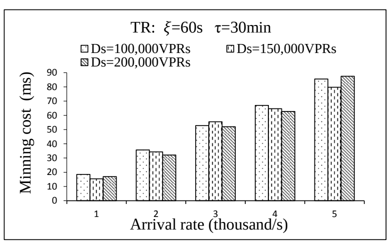
(b) Comparison of mining cost (Twitter)



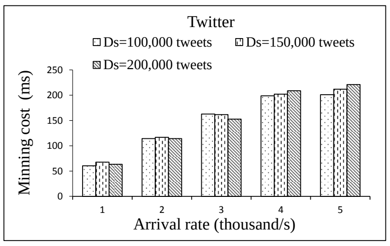
(c) Comparison of total cost (TR)



(d) Comparison of total cost (Twitter)



(e) Mining cost w.r.t D_s



(f) Mining cost w.r.t D_s

Figure 6: Performance of mining algorithms

Figure 5(d) shows that the parameter ξ has little influence on the maintenance time of the Seg-tree, but it affects the maintenance cost of DI-Index and *Matrix* more significantly. The reason is that the sizes of segments will become larger when ξ takes greater values, and the larger segments have very slight influence on the maintenance time of the Seg-tree because each segment can be inserted and removed in its entirety regardless of its length. However, the larger segments contain more objects, which can produce more elements for DI-Index and *Matrix*, and they thus need much more time to maintain the larger segments.

Fig. 5(e) shows the maintenance cost of the three index structures for the Twitter dataset. Similar to the case of the TR dataset, the Seg-tree has the less maintenance cost than DI-Index and *Matrix*.

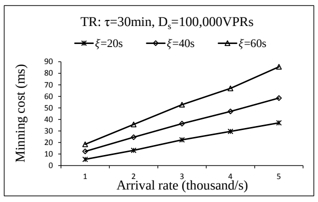
According to the aforementioned experimental results, we conclude that the Seg-tree and the DI-Index outperform the *Matrix* w.r.t. memory consumption and maintenance cost. In most cases, the performance of Seg-tree is better than that of the DI-Index structure except the memory consumption for processing the Twitter dataset.

Compression ratio. To help us better understand the memory consumption, assuming that the original data to be indexed is d_1 and the real data stored in the Seg-tree is d_2, the compression ratio of the Seg-Tree is defined as (d_1-d_2)/d_1. In Fig. 5(f), the compression ratio based on TR dataset is very high, which means the Seg-tree can compact the overlapping segments very well. As to the Twitter dataset, the compression ratio becomes very low as there are less overlapping between segments. Therefore, it would be helpful to look at the degree of overlapping between segments before deciding on the index structure for mining FCP.

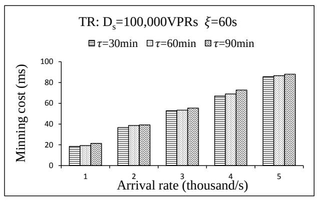
6.4 Performance comparison of the mining algorithms

Comparison of mining cost. First, we compare the mining performance of CooMine, DIMine, and MatrixMine algorithms on the two datasets in Fig. 6(a) and 6(b). For the TR dataset, the mining time of the three algorithms is almost identical. However, the mining cost of the CooMine algorithm is larger than that of the DIMine and MatrixMine methods on the Twitter dataset. This is again due to the lower degree of overlapping between segments, which renders some optimizations of the CooMine algorithm not applicable in reduing the mining cost.

Comparison of total cost. In this group of experiments, we evaluate the total cost of the three algorithms for processing the data within one second with varying arrival rates based on two datasets, and the results are shown in Fig. 6(c) and 6(d). For the set of data being processed, the total cost includes the time of inserting this set of data into the index structure and removing the relevant obsolete data from the index structure, as well as the time of mining FCPs from this set of data. The results show that CooMine performs best, and both CooMine and DIMine outperform the MatrixMine approach dramatically regardless of the dataset.



(a) Mining cost w.r.t ξ



(b) Mining cost w.r.t τ

Figure 7: FCPs w.r.t D_s

6.5 Evaluation of the CooMine algorithm

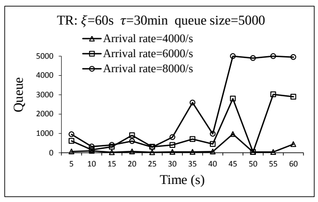
Since the CooMine algorithm is better than the other two methods with respect to the total cost, we only evaluate the influence of varying parameters on its performance.

Mining cost w.r.t. D_s. We test the influence of D_s on the performance of CooMine for mining FCPs on two datasets in Fig. 6(e) and 6(f). The results show that D_s has no evident effect on the mining cost because the CooMine algorithm only needs to search a small portion of the data to find FCPs.

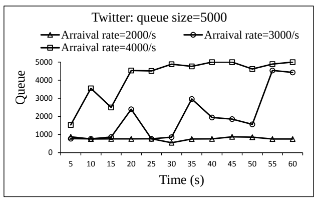
Mining cost w.r.t. ξ and τ. Fig. 7(a) shows that the mining time is also affected by the parameter ξ. The larger value of ξ will give rise to longer segments, and the segments with larger sizes will cause more LCPs between each new segment and existing ones, which can increase the mining time. Fig. 7(b) demonstrates that the parameter τ has little impact on the mining cost. This is because although the larger value of τ can cause more valid segments, the CooMine algorithm can still efficiently narrow down the search scope.

Maximum sustainable workload. To evaluate the maximum sustainable workload of the CooMine algorithm, we introduce a buffer queue with 5000 storage units to cache the incoming data, and the CooMine algorithm will fetch the data from this queue. In this case, the usage rate of the buffer queue reflects the processing capacity of the CooMine algorithm.

In Fig. 8(a), we evaluate the buffer queue usage at different time points based on the TR dataset. When the arrival rate reaches 8000 VPRs per second, the maximum usage of the buffer queue is 5000. Therefore, the maximum sustainable workload of the CooMine algorithm based on the TR dataset is 8000 VPRs per second. Fig.



(a) Maximum sustainable workload (TR)



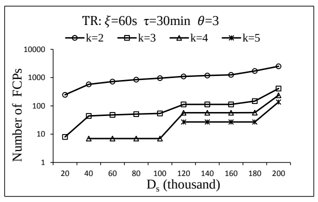
(b) Maximum sustainable workload (Twitter)

Figure 8: FCPs w.r.t D_s

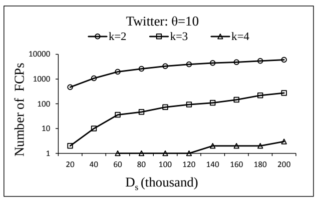
8(b) shows that the maximum sustainable workload based on the Twitter dataset is about 4000 tweets per second.

6.6 Effect of parameters on the number of FCPs

We now evaluate the influence of the parameters D_s and θ on the number of FCPs generated for the two datasets.



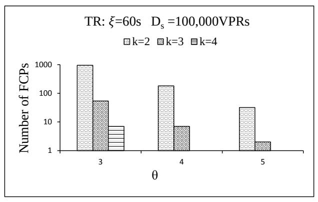
(a) Results on TR



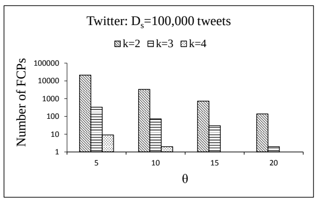
(b) Results on Twitter

Figure 9: The number of FCPs discovered w.r.t D_s

Fig. 9(a) and 9(b) show that the number of FCPs increases with more data being mined for the TR and Twitter datasets respectively. For a fixed volume of data, there exist more FCPs with smaller sizes (k).



(a) Results on TR



(b) Results on Twitter

Figure 10: The number of FCPs discovered w.r.t θ

We also test the effect of the parameter θ on the number of FCPs in Figure 10(a) and 10(b). When the value of θ becomes larger, the number of FCPs drops sharply, which coincides with our intuition that the higher the threshold, the less FCPs will appear.

Finally, we analyze the FCPs from the Twitter dataset with θ equal to 60 and illustrate some typical hot events that the FCPs imply in Table 3 and Table 4, demonstrating that mining FCP is indeed useful in such applications.

6.7 Discussion

We have compared the CooMine, DIMine, and MatrixMine approaches with respect to the memory consumption, the maintenance cost, the mining cost, and the total cost on two datasets. The results demonstrate that CooMine and DI-Mine approaches outperform the MatrixMine method significantly for mining FCPs. Between the CooMine and DI-Mine approaches, we find that the CooMine approach based on the Seg-tree is better suited to process

Table 3: Typical FCPs from the Twitter dataset

FCPs	The number of streams	Hot event
super bowl	1378	event1
green bay packers	213	
win steelers	226	
jack lalanne dies	139	event2
airport killed	111	event3
airport domodedovo	101	
union state address	409	event4
obama sotu	456	
science fair	63	
health care	261	

Table 4: Hot events

Event	Meaning
event1	*Green Bay Packers and Pittsburgh Steelers played the Super Bowl on February 6, 2011.*
event2	*Jack lalanne, the American exerciser, and nutritional expert, died on January 23, 2011.*
event3	*The Domodedovo International Airport bombing on January 24, 2011.*
event4	*Barack Obama presented the 2011 State of the Union Address on January 25, 2011.*

data streams that have much overlap between segments, while the DIMine approach is more suitable for handling data streams without overlapping segments.

7. CONCLUSION

Mining frequent co-occurrence patterns (FCPs) across multiple data streams is essential to many real-world applications, but this problem has not been addressed by exiting works. In this paper, we design the DIMine and CooMine approaches to mine FCPs using only one pass over the data streams. In both approaches, we first divide each data stream into overlapping segments, and then mine the FCPs within those segments. The DIMine approach uses an inverted index (DI-Index) to maintain the valid segments in main memory and adopts an Apriori-style heuristic to iteratively discover FCPs based on this index. In the CooMine approach, we construct the Seg-tree, a memory-based index structure, to compactly index all valid segments. Based on the Seg-tree, the CooMine approach first finds the largest common CPs between each new segment and the existing ones to narrow down the search scope, and then discover the FCPs from the obtained common CPs. Finally, we introduce a baseline method and conduct extensive experiments to compare our proposed approaches with this baseline method. The experimental results demonstrate that our proposed approaches outperform the baseline method by a significant margin.

For future work, we would like to study how to extend the proposed approaches to a distributed environment to handle greater scales of data streams, when a single machine is no longer capable of managing the large volumes of data and computation.

8. ACKNOWLEDGMENT

This work was supported in part by the 973 Program (2015CB352 500), the National Natural Science Foundation of China Grant (6127 2092), the Shandong Provincial Natural Science Foundation Grant (ZR2012 FZ004), the Science and Technology Development Program of Shandong Province (2014GGE27178), the Independent Innovation Foun-dation of Shandong University (2012ZD012), the Taishan Scholars Program, and NSERC Discovery Grants.

9. REFERENCES

[1] Trie. http://en.wikipedia.org/wiki/Trie.

[2] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. In *SIGMOD*, pages 207–216, 1993.

[3] R. Agrawal, R. Srikant, et al. Fast algorithms for mining association rules. In *VLDB*, pages 487–499, 1994.

[4] J. H. Chang and W. S. Lee. Finding recent frequent itemsets adaptively over online data streams. In *SIGKDD*, pages 487–492, 2003.

[5] Z. Chen, H. T. Shen, and X. Zhou. Discovering popular routes from trajectories. In *ICDE*, pages 900–911, 2011.

[6] C. Giannella, J. Han, J. Pei, X. Yan, and P. S. Yu. Mining frequent patterns in data streams at multiple time granularities. *Next generation data mining*, pages 191–212, 2003.

[7] F. Giannotti, M. Nanni, F. Pinelli, and D. Pedreschi. Trajectory pattern mining. In *SIGKDD*, pages 330–339, 2007.

[8] J. Guo, P. Zhang, J. Tan, and L. Guo. Mining frequent patterns across multiple data streams. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 2325–2328, 2011.

[9] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD*, pages 1–12, 2000.

[10] R. M. Karp, S. Shenker, and C. H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *TODS*, pages 51–55, 2003.

[11] J.-G. Lee, J. Han, and X. Li. Trajectory outlier detection: A partition-and-detect framework. In *ICDE*, pages 140–149, 2008.

[12] C.-S. Leung and Q. I. Khan. Dstree: a tree structure for the mining of frequent sets from data streams. In *ICDM*, pages 928–932, 2006.

[13] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD*, pages 39–44, 2005.

[14] Q. Li, Y. Zheng, X. Xie, Y. Chen, W. Liu, and W.-Y. Ma. Mining user similarity based on location history. In *SIGSPATIAL*, pages 34–45, 2008.

[15] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *VLDB*, pages 346–357, 2002.

[16] A. Monreale, F. Pinelli, R. Trasarti, and F. Giannotti. Wherenext: a location predictor on trajectory pattern mining. In *SIGKDD*, pages 637–646, 2009.

[17] B. Mozafari, H. Thakkar, and C. Zaniolo. Verifying and mining frequent patterns from large windows over data streams. In *ICDE*, pages 179–188, 2008.

[18] R. T. Ng, L. V. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *SIGMOD*, pages 13–24, 1998.

[19] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: Alternatives and implications. In *SIGMOD*, pages 343–354, 1998.

[20] S. K. Tanbeer, C. F. Ahmed, B.-S. Jeong, and Y.-K. Lee. Efficient single-pass frequent pattern mining using a prefix-tree. *Information Sciences*, 179(5):559–583, 2009.

[21] J. X. Yu, Z. Chong, H. Lu, and A. Zhou. False positive or false negative: mining frequent itemsets from high speed transactional data streams. In *VLDB*, pages 204–215, 2004.

Extracting Aggregate Answer Statistics for Integration

Zainab Zolaktaf
University of British Columbia
Vancouver, BC, Canada
zolaktaf@cs.ubc.ca

Jian Xu
Microsoft Corporation
Redmond, WA, USA
xujian@microsoft.com

Rachel Pottinger
University of British Columbia
Vancouver, BC, Canada
rap@cs.ubc.ca

ABSTRACT

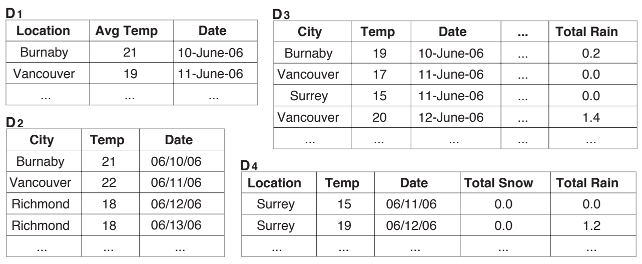
Aggregate queries in integration contexts often do not have one "true" answer; there can be multiple correct answers for the same aggregate query. This is due to the existence of duplicate or overlapping data points, possibly with different values, across the data sources. Depending on the choice of data source combinations that are used to answer the query, different answers can be generated. Thus, representing the answer to the aggregate query as an *answer distribution* instead of a single scalar value, will allow the users to better understand the range of possible answers.

This work provides a suite of methods for extracting statistics that convey meaningful information about aggregate query answers in heterogeneous integration settings. We focus on the following challenges: 1. determining which statistics best represent an answer's distribution; and 2. efficiently computing the desired statistics.

Our solution includes the following answer statistics 1. a set of *point estimates* with confidence intervals; 2. a *high coverage interval* that unveils "hot areas" in a distribution; and 3. a *stability score* that measures the impact of source dynamics. We optimize the extraction of the above statistical information by minimizing the sampling load and applying fast approximate algorithms. We verify the effectiveness and efficiency of our methods with empirical studies using real-life and synthetic, scaled data sets.

1. INTRODUCTION

Aggregate queries are fundamental to relational databases. They group sets of data values and calculate informative statistics such as average, median, and sum. They are also important in heterogeneous integration systems [1, 2, 9] where the focus shifts from querying a single database to querying multiple, independently managed, domain heterogeneous databases. The characteristics of heterogeneous information systems make the generation of meaningful aggregate values for heterogeneous information systems significantly more challenging than aggregations in a typical



D1

Location	Avg Temp	Date
Burnaby	21	10-June-06
Vancouver	19	11-June-06
...	...	...

D2

City	Temp	Date
Burnaby	21	06/10/06
Vancouver	22	06/11/06
Richmond	18	06/12/06
Richmond	18	06/13/06
...	...	...

D3

City	Temp	Date	...	Total Rain
Burnaby	19	10-June-06	...	0.2
Vancouver	17	11-June-06	...	0.0
Surrey	15	11-June-06	...	0.0
Vancouver	20	12-June-06	...	1.4
...	...	...	...	...

D4

Location	Temp	Date	Total Snow	Total Rain
Surrey	15	06/11/06	0.0	0.0
Surrey	19	06/12/06	0.0	1.2
...	...	...	...	...

Figure 1: Climate data from BC weather stations.

relational database.

Answering aggregate queries in a heterogeneous information system often requires combining sets of data that are segmented across multiple sources. These sources may vary substantially with regard to their schemas and the instances they hold, i.e., semantically related content may be stored in different structures, at different levels of granularity, in different representations, and multiple formats.

There are three levels of heterogeneity in heterogeneous information systems [19]. The first is schema-level. This occurs when there are different schema elements representing the same concept, e.g., one schema may contain a "temp" attribute, while another contains a "temperature" attribute. The second is instance-level heterogeneity. This level requires performing entity resolution to tell if two objects are the same, e.g., that the data for "Vancouver Weather 2006/06/11" in one data source is the same as "Vancouver Weather 06/11/2006" in another. The third is value-level heterogeneity. Heterogeneity at this level deals with the fact that because the sources are independently created and maintained, a given data point can have multiple, inconsistent values across the sources. For example, one source may have the high temperature for Vancouver on 06/11/2006 as 17C, while another may list it as 19C. It is this value-level heterogeneity with which we are concerned throughout this paper. Particularly, we look at the problem of how to handle value-level heterogeneity in aggregations.

To illustrate the problem, consider JIIRP [17], a real-world disaster management project. In JIIRP, data from various sources are combined to simulate the impact of natural disasters. For example, JIIRP assesses weather phenomena and climate data to help plan emergency responses. Figure 1 shows local data sources containing climate data for cities located in British Columbia (BC, Canada). As shown in the figure, the sources differ in terms of their coverage of

10.5441/002/edbt.2015.09

the data instances and attributes. Additionally, data sources D_1, D_2, and D_3 hold different values for the same data point Vancouver on 06/11/2006.

Next, consider the following query, in which the data sources are queried to find the average temperature for months with an average temperature above 20 degrees Celsius:

```
Select Average(Temp), Month(Date), Province(Location)
From SemIS
GROUP BY Province(Location), Month(Date)
HAVING Average(Temp) > 20
```

Applying standard aggregation, however, is incorrect. In particular, standard aggregation would simply return the average of all the points. This is problematic for two reasons: (1) There are some values which are present in more than one source (e.g., Vancouver's temperature on 06/11/2006 is represented in all of sources D_1, D_2, and D_3; taking the simple average will cause Vancouver to be over-represented in the average.) (2) The different sources may have different values for the same conceptual answer (e.g., depending on which source is used, the temperature for Vancouver on 06/11/2006 is either 17, 19, or 22).

The correct aggregation requires using only one value per data point:

$$Average(t) = \frac{\sum_{c=1}^{|\mathcal{C}_{BC}|} \sum_{d=1}^{|\mathcal{D}_m|} t_{c,d}}{|\mathcal{C}_{BC}| * |\mathcal{D}_m|} \tag{1.1}$$

where t is temperature, c represents city, $|\mathcal{C}_{BC}|$ is the number of cities in BC, d represents day, $|\mathcal{D}_m|$ is the number of days in month m. The above aggregation requires 1470 data points (49 cities in BC * 30 days), each of which could have several duplicates across the sources. Depending on the choice of source and value combinations, there can be a whole range of *viable answers*.

For such queries, enumerating all the possible value combinations, and generating the entire set of viable answers as the answer to the aggregate, not only does not scale well (due to combinatorial explosion with regard to the number of data points and possible duplicates), but would still require the users to analyze the results and determine suitable answers.

This problem is not unique to weather data or the JIIRP scenario, for example, [19] examined inconsistency and redundancy at the value-level, in the stock and flight domains on the Deep Web. However, in [19] the authors assumed that there was a single "true" answer, which we do not.

In this paper, we assume meta-information that describes the mappings and bindings between data sources is available [25]. Similar to [19], we focus on value-level heterogeneity. Specifically, we propose a suite of methods for summarizing aggregate query answers in integration settings.

Our previous work [25] created a system where one could ask such aggregate queries. It defined what the possible viable answers were, but then it randomly chose a viable answer. This is inappropriate both since it does not explain the choice, and the aggregate answer would fluctuate upon reprocessing the query. Selecting a best guess answer or a top-K answer set is also inadequate; there is often no global mediator available to choose the value and source combination for the aggregation.

Our solution consists of first estimating the distribution of the viable answers. However, instead of enumerating all the viable answers and computing the full, accurate distribution, we combine a variety of statistical estimations into what we term the *viable answer distribution.* We then efficiently extract statistical summaries of the viable answer distribution to allow the user to better interpret and understand the viable answers. Thus, we contribute the following:

- We define aggregate answers in heterogeneous information systems as a distribution of viable answers computed from different data source and value combinations.

- We provide three summary statistics for the viable answer distribution, consisting of: 1. key point statistics with user defined confidence intervals; 2. high coverage intervals to convey distribution shape information and 3. a stability measure for the aggregation.

- We provide algorithms to efficiently extract the above statistics, including 1. estimating point statistics using sampling and ways to reduce sampling overhead while keeping the confident intervals tight; 2. a fast, greedy algorithm to extract hot areas; and 3. using a probabilistic model to calculate stability scores without simulating source removal. Overall, we can support online extraction of aggregation statistics.

- We describe our empirical study using real-life and synthetic data sets, further verifying our theoretical and algorithmic claims on effectiveness and efficiency.

The paper is organized as follows. We describe preliminary statistical methods applied in our solution in Section 2 and formalize the problem in Section 3. We present the technical details of the distribution estimate process and optimizations in Section 4, the empirical study in Section 5, and review related work in Section 6. We conclude in Section 7.

2. PRELIMINARIES

In this work we use lower case letters for scalars (e.g., a) and typeset the sets (e.g., $\mathcal{A}$); all other variables, including random variables, are denoted by capital letters (e.g., A).

2.1 Bootstrap sampling and bagging

Bootstrap sampling, or bootstrapping, is a resampling technique which combined with Bootstrap aggregating (bagging) [7], can be used to improve the quality of an estimate. The bootstrapping starts with an initial (small) sample set $\mathcal{S}_{alg}$ sampled according to some sampling algorithm alg. This set is then resampled according to alg to obtain a set of bootstrapped sample sets $\mathcal{S}_{boot} = \{\mathcal{B}^i_{boot}\}$ where $i = 1, 2, \ldots, |\mathcal{S}_{boot}|$. Next, an estimator is applied to each set to obtain an ensemble of bootstrapped estimates $\mathcal{E}_{boot} = \{E^i_{boot}\}$. Bagging then approximates a more accurate estimate with tighter confidence intervals by combining (e.g., averaging) this ensemble of estimates. For example, the median value of $\mathcal{E}_{boot}$ can be used as the estimate of the mean of the distribution. We use the above methods to estimate the density of the viable answer distribution.

2.2 Kernel density estimation

Kernel density estimation (KDE) estimates the probability density function (pdf) of a distribution using a sample set drawn from the distribution. We use kernel rather than histogram density estimation due to properties such as smoothness, independence of parameters like bin size, and because KDE often converges to the true density faster. It works as follows: Let the sample set be $\mathcal{S}_{alg} = \{x_i\}$, where $i = 1, 2, \ldots, |\mathcal{S}_{alg}|$. A sample point x_i's contribution to the pdf is measured using a kernel function $K(\frac{x-x_i}{h})$, where h is the bandwidth. After kernels are applied to all x_i's, the pdf is estimated by $f(x) = \frac{1}{|\mathcal{S}_{alg}|h} \sum_1^{|\mathcal{S}_{alg}|} K(\frac{x-x_i}{h})$.

Among the various possible kernel functions, typically Gaussian kernels $K(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$, are used for convenience of theoretical analysis. Note that using a Gaussian kernel makes no assumption that the data adheres to a Gaussian distribution. The bandwidth parameter h controls the localness of a point's impact on the distribution. A large h results in a smooth density function, but is likely to underfit, whereas a small h fits better on the sample points but is likely to over-fit. Selecting an appropriate h value is challenging but there are automatic methods for choosing the value of h [6]. We use KDE to estimate the viable answer density distribution.

2.3 Distance measures for distributions

Several distance measures exist for comparing and quantifying the difference between two distributions p and q. For example, $d_{L_2}(p, q) = \sqrt{\int [p(x) - q(x)]^2 \, dx}$ is the L_2 norm. Also, $d_{Bh}(p, q) = \int \sqrt{p(x)\, q(x)} \, dx$ is the Bhattacharyya distance measure [4] that uses the integral of point-wise product of the two distributions.

In Section 4.4, we quantify the changes of the viable answer distribution under different data source settings. As we will see, the complexity of computing a distance measure largely depends on the mathematical properties of the measure. Our analysis shows that stability scores measured by d_{L_2} and d_{Bh}, as defined above, can be computed efficiently.

3. PROBLEM FORMALIZATION

Data sources in a heterogeneous information system may vary significantly in terms of coverage, quality, and accuracy. Regarding coverage, often a single data source provides information about a subset of the instances and a subset of the object attributes. Therefore, data values relevant to an aggregate query can be segmented across multiple sources in a heterogeneous information system. Furthermore, as in [19], the data sources can contain heterogeneity at three levels: schema-level, instance-level, and value-level. Heterogeneity at the schema-level and instance-levels means semantically related content can be stored using different schema elements and structures, and represented by different instances. At the value-level, depending on the source quality and accuracy, the sources can have inconsistent, or even conflicting data values for the same data points. In this work, similar to [19], we focus on value-level heterogeneity. We assume meta-information that describes the mappings and bindings between data sources is available [25].

Figure 1 demonstrated an example scenario with four data sources $D_1, \ldots, D_4$, and their corresponding data instances. As shown in the figure, the sources held different values for the same data point, e.g., Vancouver on 06/11/2006. As a consequence of the data value overlaps and inconsistencies, the answer to aggregate queries, such as "Sum(Temp)" for specific date ranges and locations, depends on the combination of data sources and instances that are selected. Therefore, the answer to the aggregation is a distribution of values, rather than a single scalar value. In order to show which values are under consideration as answers, we use the term "viable answer". While the work in this paper does not depend on the definition, for concreteness, we adopt the definition from [25]:

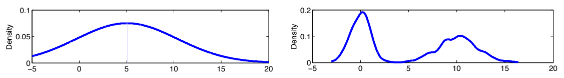


Figure 2: Two distributions with the same mean (5.0) and variance (5.0), but different shapes.

DEFINITION 1. *[Viable answer]*
Let $\mathcal{D}$ denote a set of data sources answering an aggregation, and let $v = agg(\mathcal{Z})$ be the aggregated value computed from some $\mathcal{Z} \subseteq \mathcal{D}$. Let $\mathcal{V} = \{v_i\}$ be the set of aggregated values from all possible source combinations. A viable answer to an aggregation is a value in the interval $W = [inf(\mathcal{V}), sup(\mathcal{V})]$ that adheres to type restrictions (e.g., integer) where inf and sup are infimum (greatest lower bound) and supremum (smallest upper bound), respectively. $\square$

Note that this definition allows any value in the defined interval, even if it does not correspond to the value produced by any source combination [25]. Furthermore, we assume prior knowledge regarding the coverage, quality and accuracy of the data sources is not available. Therefore, the data sources selected for inclusion have equal importance, but their contribution to the aggregate answer may not be of equal amount. As a reminder, in [25] we randomly chose a viable answer, which is inappropriate, both because it does not explain the choice and the aggregate answer would fluctuate upon reprocessing the query.

DEFINITION 2. *[Viable answer distribution] Let the random variable X be the answer to an aggregate query whose pdf is $f_X^{\mathcal{D}}(x) : W \to \mathbb{R}$. In this notation, the superscript $\mathcal{D}$ denotes the source set used to compute the viable answer set. We refer to $f_X^{\mathcal{D}}$ as the viable answer distribution.*
$\square$

Our objective, in this work, is to efficiently sample the set of viable answers, estimate a viable answer distribution, and report informative statistical summaries that allow the user to better understand the range of viable answers. Typically, the range of possible query answers are conveyed using point estimates such as mean and variance. However, such statistics are not informative when the shape of the density function is unknown; distributions with different shapes can have the same mean and variance but deliver very different information (depicted in Figure 2).

We propose to use the following three statistics as summaries of the viable answer distribution, and the answer to aggregate queries in heterogeneous information systems:

1. **Key point statistics:** Include mean, variance, and skewness of the viable answer distribution. These help users to identify appropriate scalar values for aggregate answers.
2. **High coverage intervals:** Tell where the majority of viable answers can be found. This is particularly useful when the distribution is multi-modal (i.e., the distribution has a pdf with two or more significant peaks.)

3. **A stability measure:** Tells how much the viable answer distribution would change when updates happen or some data sources become unavailable. It helps the heterogeneous information system to decide if a re-processing of an aggregate query is necessary.

The above three statistics communicate semantic information that enable the user to easily interpret and understand the distribution. Specifically, the mean, variance and skewness are standard measures most often desired in describing a distribution. High coverage intervals fill the gap that the point statistics are incapable of providing: "shape" information about a distribution, which is especially useful when the distribution is multi-modal. While the static behavior of the answer distribution is described by the first two statistics, the stability measure captures the update behavior of the distribution.

Furthermore, to focus our work, we assume that the queries that are being asked are aggregate queries. Specifically, we consider sum, average, median, variance, and standard deviation. While our methods may work on other aggregate functions, they were not our focus. We leave removing this restriction as future work.

Algorithm 1: Overall algorithm for extracting statistics

```
input  : (User specified) Query $Q$, Data sources $\mathcal{D}$,
         Confidence level $1-\alpha$, Desired coverage $\theta$.
input  : (System parameters) Query processor $QP$,
         Initial sample size $|\mathcal{S}_{uniS}|$ (400), #bootstrap
         sample sets $|\mathcal{S}_{boot}|$ (50), Bootstrap sample size
         $|\mathcal{B}^i_{boot}|$(400), Distance measure $d$ ($d_{L_2}$).
output : Mean, variance, skewness point estimates
         $(\mu, \sigma^2, \gamma_1)$, confidence intervals
         $(CI_\mu, CI_{\sigma^2}, CI_{\gamma_1})$;
output : High coverage intervals $(\mathcal{I}, L, C)$;
output : Stability score for $\mathcal{D}$ $Stab_d$.
1  begin
       // Unbiased sampling on viable answers.
2      $\mathcal{S}_{uniS} = UniS(QP(Q), \mathcal{D}, |\mathcal{S}_{uniS}|)$;
       // Bootstrap Sampling.
3      $\mathcal{S}_{boot} = BootstrapSampling(\mathcal{S}_{uniS}, |\mathcal{S}_{boot}|, |\mathcal{B}^i_{boot}|)$;
       // Estimate point statistics.
4      $(\mu, \sigma^2, \gamma_1) = EstPointStatistics(\mathcal{S}_{boot})$;
       // Estimate confidence intervals.
5      $(CI_\mu, CI_{\sigma^2}, CI_{\gamma_1}) = EstCI(\mathcal{S}_{boot}, (\mu, \sigma^2, \gamma_1), \alpha)$;
       // Estimate density function using KDE.
6      $f^{\mathcal{D}}_X = EstDensityFunction(\mathcal{S}_{boot})$;
       // Obtain high coverage intervals.
7      $(\mathcal{I}, L, C)$= $GreedyAlgorithmCIO(f^{\mathcal{D}}_X, \theta)$;
       // Obtain stability score.
8      $Stab_d = Stab(f^{\mathcal{D}}_X, d)$;
9      return Obtained viable answer statistics.
10 end
```

4. EXTRACTING ANSWER STATISTICS

4.1 Overview

Algorithm 1 describes the overall procedure for extracting statistical summaries of the viable answer distribution. The extraction of all three statistics share the uniS sampling and bootstrap sampling steps. In the uniS sampling step (line 2) a set of viable answers are sampled by processing the aggregate query using values from different data sources. This is the most expensive step. The answer set is then bootstrap resampled (line 3). This enables the computation of point statistics such as mean, variance and skewness (line 4) with confidence intervals (line 5). The density estimation process (line 6) is used for finding high coverage intervals (line 7) and for stability analysis (line 8). We perform density estimation on sample sets rather than the entire data set to ensure that the statistics extraction scales well.

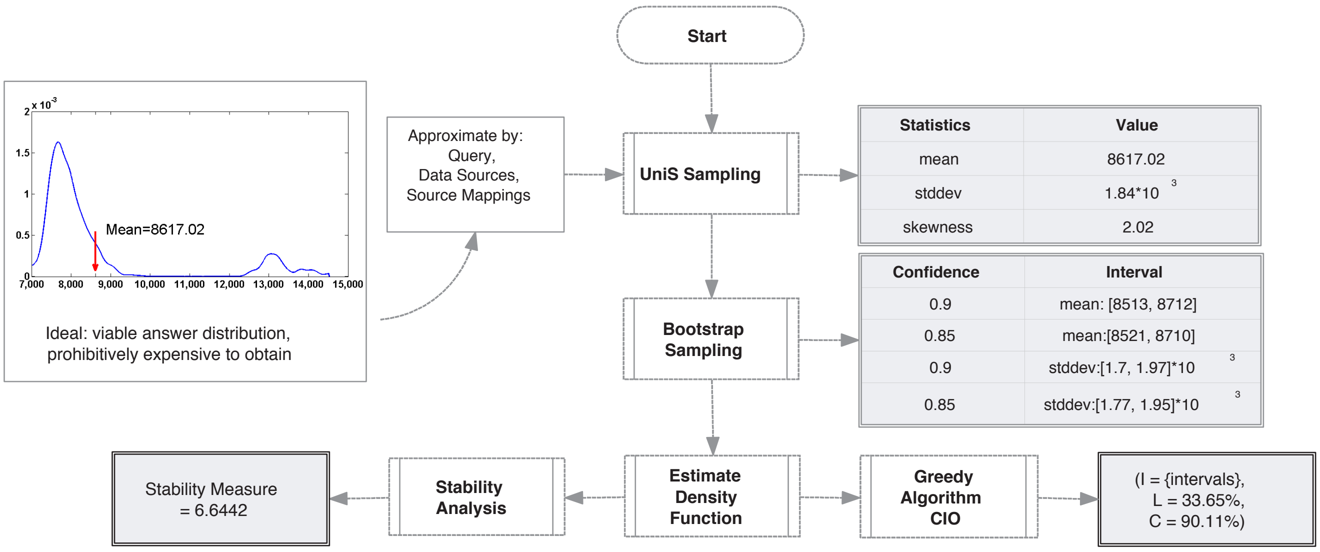
Figure 3 shows the application of Algorithm 1 to obtain answer statistics for the aggregation "Sum(Temp)" over the data sources in Figure 1. Ideally we would prefer to have the exact target viable answer distribution (shown in the top left corner of the figure), however it does not scale well to get this exact distribution. Instead we approximate using the inputs, consisting of the query, data sources, and the mappings between the sources. The algorithm works as follows: UniS sampling samples the data sources to obtain $\mathcal{S}_{uniS}$, a set of viable answer samples, where $|\mathcal{S}_{uniS}| = 400$. Next, this set is resampled to obtain a set of bootstrap sample sets $\mathcal{S}_{boot} = \{\mathcal{B}^i_{boot}\}$, where $|\mathcal{S}_{boot}| = 50$ and $|\mathcal{B}^i_{boot}| = 400$, which are then used to help obtain the 90% and 85% confidence intervals for mean and standard deviation (stddev). After the density function is estimated, Greedy algorithm CIO (Algorithm 2) is used to obtain 3 intervals covering 34% of the range of values, and approximately 90% of the estimated probability distribution. Finally, a stability score of 6.6442 is computed for the query. The outputs of the algorithm are shaded in grey.

In summary, the proposed methods efficiently provide the user with answer statistics that simplify the interpretation of the range of viable answers. This is in contrast to inefficient methods that not only do not scale well, but also require the user to examine and interpret the answers. Sections 4.2 – 4.4 , describe how the three statistics are extracted in detail.

4.2 Sampling and point statistics

We use the term *component* to indicate a data point that an aggregate requires, e.g., in the climate data example in the introduction, a component would be the temperature for Vancouver on 06/11/2006. The process of sampling a viable answer is: (1) find an assignment that determines the use of values from data sources and (2) compute the answer using the chosen assignments. We do not assume prior knowledge regarding the quality, reliability, accuracy, and coverage of the sources. In this context, to ensure correctness, the sampling procedure must select the data sources uniformly and independently to contribute to the aggregate.

We designed a sampling scheme called *uniS* that satisfies the above requirements. Let $\mathcal{D} = \{D_i\}$ where $i = 1, 2, \ldots, |\mathcal{D}|$ denote the data sources, $\mathcal{C}_i$ be the set of components on D_i, and $\mathcal{C}$ be the set of all components needed by the aggregate. UniS starts with an empty component set $\mathcal{T}_0$ and an initial partial aggregate p_0. Step i of uniS uniformly selects one data source D_k, and attempts to add as many components in $\mathcal{C}_k$ to the aggregate, i.e., it updates $\mathcal{T}_i = \mathcal{T}_{i-1} \cup \mathcal{C}_k$ and p_i with partial aggregate computed over component set $\mathcal{A}_i = \{c \mid c \in \mathcal{C}_k, c \notin \mathcal{T}_{i-1}\}$. This process is repeated until $\mathcal{T} = \mathcal{C}$ or all n data sources have been visited. It then computes a final aggregate from the partial


Start
2 x 10^-3
1.5
1
0.5
0
7,000
8,000
9,000
10,000
11,000
12,000
13,000
14,000
15,000
Mean=8617.02
Ideal: viable answer distribution, prohibitively expensive to obtain
Approximate by: Query, Data Sources, Source Mappings
UniS Sampling
Bootstrap Sampling
Estimate Density Function
Greedy Algorithm CIO
Stability Analysis
Stability Measure = 6.6442
(I = {intervals}, L = 33.65%, C = 90.11%)

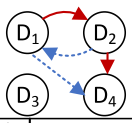
Statistics	Value
mean	8617.02
stddev	$1.84*10^3$
skewness	2.02

Confidence	Interval
0.9	mean: [8513, 8712]
0.85	mean:[8521, 8710]
0.9	stddev:$[1.7, 1.97]*10^3$
0.85	stddev:$[1.77, 1.95]*10^3$

Figure 3: Application of Algorithm 1 to extract statistics for "Sum(Temp)" over data sources in Figure 1.

aggregate and uses it as a viable answer sample. [1]

Figure 4 shows an example scenario where uniS sampling is applied to the sources in Figure 1 for an avg() aggregate over $\mathcal{C} = \{c_1, \ldots, c_5\}$. It shows two different selection paths, $path(D_1, D_2, D_4)$ and $path(D_2, D_1, D_4)$. $\mathcal{T}_i$ is the set of covered components, p_i is the incrementally maintained partial aggregate. The arrows show the different selection paths for $\mathcal{T}_i$ and p_i. For $path(D_1, D_2, D_4)$, the algorithm begins at node D_1 and uses all the components in D_1. The remaining two components c_4 and c_5 are sampled from sources D_2 and D_4, respectively. Note that same component, e.g, c_1, can have different values on different data sources. Hence, using an alternative path, $path(D_2, D_1, D_4)$, yields a different viable answer; since D_2 is visited first, uniS takes both components, c_1 and c_4, from D_2.



Node	Components
D_1	$\{c_1, c_2, c_3\}$
D_2	$\{c_1, c_4\}$
D_4	$\{c_1, c_5\}$

Node	path(D_1, D_2, D_4) →
D_1	$T_1=\{c_1, c_2, c_3\}$ $p_1=g(T_1)$
D_2	$T_2=T_1\cup\{c_1, c_4\}$ $p_2=g(p_1, \{c_4\})$
D_4	$T_3=T_2\cup\{c_1, c_5\}$ $p_3=g(p_2,\{c_5\})$

Node	path(D_2, D_1, D_4) ⇢
D_1	$T_1=\{c_1, c_4\}$ $p_1=g(T_1)$
D_2	$T_2=T_1\cup\{c_1,c_2,c_3\}$ $p_2=g(p_1, \{c_2,c_3\})$
D_4	$T_3=T_2\cup\{c_1, c_5\}$ $p_3=g(p_2,\{c_5\})$

Figure 4: UniS sampling for selection paths of $\{\mathbf{D_1}, \mathbf{D_2}, \mathbf{D_4}\}$ and $\{\mathbf{D_2}, \mathbf{D_1}, \mathbf{D_4}\}$, where $\mathbf{g} = \mathbf{sum}()$, and $\mathbf{p_i}$ is the incrementally maintained partial aggregate.

Drawing a sample from possibly distributed data sources involves processing an aggregate query with a randomly selected value assignment. Although partial-final aggregates helps to distribute the computational load of each aggregation, applying uniS to draw samples is still a costly operation. Thus, it is desirable to minimize $|\mathcal{S}_{uniS}|$. To this end, we apply bootstrap resampling on the sampled viable answers to improve the confidence for point statistics such as mean and variance. The three parameters: confidence level $1 - \alpha$, confidence interval length denoted by $len(CI)$, and the sampling size $|\mathcal{S}_{uniS}|$ are correlated to affect the computation of confidence intervals. Basically, the larger the samples in the initial set, the higher the accuracy of the point estimates and the confidence intervals. However, to reduce computational costs, we start with a fixed initial sample set and incrementally increase the size of this set (i.e., perform uniS sampling). With each increment, we perform bootstrap resampling and assess the value of $len(CI)$ with the specified α. The procedure ends when the values are satisfactory. Our implementation of bootstrapping uses the standard BC_a [13] method to obtain good quality confidence intervals using small amount of initial samples.

4.3 High coverage intervals and optimization

To better understand how viable answers are distributed, we estimate the viable answer distribution (Definition 2) using KDE (Section 2.2). Specifically, we perform density estimation for each bootstrap sample set and use the normalized point-wise mean of all the estimates as the viable answer distribution. Furthermore, we use two methods in addition to standard KDE. The adaptive method described in [6] automatically chooses the value of the bandwidth h depending on the sample set. Bagging (Section 2.1) makes use of the resampled set from bootstrapping. These methods help obtain a density estimation that is both smooth and stable, which is required to extract high coverage intervals and compute stability scores.

We now describe an algorithm for extracting statistics that convey shape information for $f_X^{\mathcal{D}}$, the pdf estimated by KDE.

DEFINITION 3. *[High coverage interval]*
Given an estimated viable answer distribution $f_X^{\mathcal{D}}$, and the viable answer range W, a high coverage interval *is a triple $(\mathcal{I}, L, C)$, where $\mathcal{I} = \{(I_i, C_i) : C_i = \int_{I_i} f_X^{\mathcal{D}}(x)dx$ and $I_i \subseteq W\}$; C_i is the coverage of I_i. $L = \frac{\sum_i |I_i|}{|W|}$ is the fraction*

[1]As an example for partial-final aggregate, for final aggregate avg(), the partial aggregate is sum().

of the intervals' total length to the viable answer range, and $C = \sum_i C_i$ is the total coverage. □

Definition 4. *[Coverage interval optimization (CIO)] Given a density function $f_X^{\mathcal{D}}$ for a distribution defined on a finite range, a coverage threshold $0 \le \theta \le 1$, and a constant t representing the number of modes, the CIO problem finds k intervals $I_1, I_2, \ldots, I_k$ where $k \le t$, to minimize $\sum_{i=1}^{k} |I_i|$ subject to $\sum_{i=1}^{k} \int_{I_i} f_X^{\mathcal{D}}(x)dx \ge \theta$.* □

We further note the following: First, the criterion of minimizing the total interval makes intuitive sense, e.g., the reported high coverage intervals, whether in the weather, the flight, or the stock exchange domain, should be as small as possible. For example, in the flight domain, it is preferable to have shorter rather than larger intervals for the departure time of a certain flight. Second, the coverage threshold θ, conveys the percentage of information covered by the data sources. The desired level can be defined by the user. Third, for single mode distributions (e.g., a Gaussian), the optimal solution is the classical $100\% * \theta$ confidence interval around the mode. The coverage intervals are more useful and deliver important information for multi-modal distributions.

Furthermore, note that k is not a given variable; the CIO problem finds k intervals to minimize the total length of the intervals returned, such that their coverage is above a threshold. Larger coverage is obtained by selecting higher modes. Motivated by this, we propose Algorithm 2 for the extraction of high coverage intervals. The inputs to the algorithm are the pdf $f_X^{\mathcal{D}}$, the desired coverage θ, and t modes of $f_X^{\mathcal{D}}$. [2] The algorithm works by greedily picking up new intervals around the modes (lines 5–7) or extending previously picked intervals (lines 9–11) for the highest $t-1$ modes. For the last mode, the interval is set to cover the average amount of the additional coverage needed (lines 17,18). The algorithm returns the obtained high coverage intervals if the desired coverage is met or it has finished searching all the t modes. Our formal analysis of the greedy algorithm relies on the following theorem.

Theorem 4.1 (CIO mode containment property). *If the probability density function $f_X^{\mathcal{D}}$ of a distribution (1) has t modes and (2) is second-order differentiable everywhere on its range, and the optimal solution for CIO has $k \le t$ intervals, then the largest k modes are contained by the k intervals in the optimal solution.*

Proof If an optimal solution for CIO has k intervals, but the i-th largest mode $(x_i, f_X^{\mathcal{D}}(x_i)), i \le k$ is not in the optimal solution, then there must exist an interval I_j for which any point $x \in I_j$ satisfies $f_X^{\mathcal{D}}(x) < f_X^{\mathcal{D}}(x_i)$; therefore we can construct a new interval around x_i and improve the previous result. □

If the conditions in Theorem 4.1 are satisfied, then the algorithm returns an optimal CIO solution. Otherwise, the greedy algorithm returns an approximation. There are two scenarios for an approximation: (1) the returned intervals may not reach the desired coverage, and (2) optimally choosing the next interval to extend coverage requires picking the I_j that has the minimal $|f_X^{\mathcal{D}\prime}(x_t^{+/-})|$ (i.e., the largest incremental on coverage).

[2]Because $f_X^{\mathcal{D}}$ is one-dimensional, the modes are easily computed numerically. We omit details on mode seeking.

Algorithm 2: Greedy algorithm CIO

```
input  : Estimated density function $f_X^{\mathcal{D}}$ over range $W$
input  : Desired coverage $\theta$
input  : A set $\mathcal{M} = \{(x_i, m_i)\}$ containing $t$ modes of $f_X^{\mathcal{D}}$
output : High coverage intervals $(\mathcal{I}, L, C)$
1  begin
2      $C \leftarrow 0.0$; $i \leftarrow 1$; array $s$; $\Omega \leftarrow \emptyset$;
3      Sort $\mathcal{M}$ by $m_i$ in descending order;
4      while $C < \theta$, $i \le (t-1)$ do
5          $x_i^- \leftarrow$ largest $x$ s.t. $x < x_i$, $f_X^{\mathcal{D}}(x) = m_{i+1}$;
6          $x_i^+ \leftarrow$ smallest $x$ s.t. $x > x_i$, $f_X^{\mathcal{D}}(x) = m_{i+1}$;
7          $s[i] \leftarrow (x_i^-, x_i^+)$; $\Omega \leftarrow \Omega \cup s[i]$; $C \leftarrow \int_\Omega f_X^{\mathcal{D}}(x)dx$;
           $j \leftarrow 1$;
8          while $C \le \theta$ do
9              $x_j^- \leftarrow$ largest $x$ s.t. $x < x_j$, $f_X^{\mathcal{D}}(x) = m_{i+1}$;
10             $x_j^+ \leftarrow$ smallest $x$ s.t. $x > x_j$, $f_X^{\mathcal{D}}(x) = m_{i+1}$;
11             $s[j] \leftarrow (x_j^-, x_j^+)$; $\Omega \leftarrow \Omega \cup s[j]$;
               $C \leftarrow \int_\Omega f_X^{\mathcal{D}}(x)dx$;
12             $j \leftarrow j+1$;
13         end
14         $i \leftarrow i+1$;
15     end
16     if $C \le \theta$ then
17         Find $(x_t^-, x_t^+)$ s.t. $x_t^- < x_t < x_t^+$ ,
           $\int_{x_t^-}^{x_t^+} f_X^{\mathcal{D}}(x)dx = \frac{1}{t}(\theta - C)$;
18         $s[t] \leftarrow (x_t^-, x_t^+)$; $\Omega \leftarrow \Omega \cup s[t]$; $C \leftarrow \int_\Omega f_X^{\mathcal{D}}(x)dx$;
19     end
20     Let $\omega_1..\omega_k$ be disjoint intervals s.t. $\bigcup_1^k \omega_i = \Omega$;
21     foreach $\omega_i$ do
22         $C_i = \int_{\omega_i} f_X^{\mathcal{D}}(x)dx$;
23     end
24     $\mathcal{I} \leftarrow \{(\omega_i, C_i)\}$; $L \leftarrow \sum_1^k |\omega_i|/|W|$;
25     return $(\mathcal{I}, L, C)$;
26 end
```

Obtaining an optimal solution in the above cases requires knowing the first derivative of the density function everywhere, which will bring a substantial cost to the computational overhead. Therefore, we decided to use an approximate answer. Another reason for not pursuing the full optimal solution is that the density function itself is an estimation, and thus has a built-in error. Our empirical study suggests that the greedy algorithm gives a good approximation, and more importantly for query processing, it is fast and scalable.

While the above CIO setting is useful in many situations, the dual of CIO is desired when we are constrained to a pre-determined interval length and asked to return the best possible coverage.

Definition 5. *[Dual of CIO] The dual problem of CIO optimizes the selection of intervals to maximize the coverage. The optimization maximizes $\sum_{i=1}^{k} \int_{I_i} f_X^{\mathcal{D}}(x)dx$ subject to $\sum_{i=1}^{k} |I_i| = \gamma$, where γ is a user-specified parameter.* □

The greedy algorithm can be easily modified for the dual of CIO by modifying the termination criteria to check if the

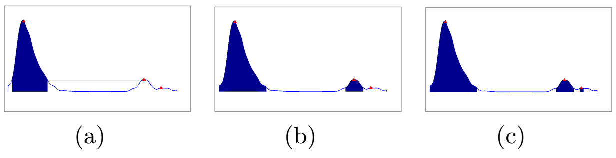

(a) (b) (c)

Figure 5: Finding high coverage intervals (Algorithm 2).

total length of the current set of intervals exceeds length γ and return the size of the covered region.

Figure 5 shows how the high coverage intervals are found for a pdf with 3 modes: the greedy algorithm starts from the highest mode and extends the coverage to lower modes until the coverage of the currently discovered intervals meets the coverage requirement. Eventually 3 intervals are reported.

The returned intervals deliver important information about the aggregation answers. For a fixed coverage θ (e.g., 0.9) and a single mode distribution, if the returned interval length is small, it means that different combinations of sources result in similar aggregation answers. It often indicates that the user can be quite confident of the returned answer. However, we still need to be careful to correctly interpret the result. A small interval length does not necessarily mean that all sources are holding the same value for the same component. It can be the case that the values of some components dominate others. For example, some components are significantly larger than others in a *sum* aggregation.

Additionally, high coverage intervals can be applied in uncertain and probabilistic databases [22]. Such databases represent an attribute as a set of value and probability pairs, $att = \{(A, Pr(A))\}$, where A represents the range of possible values and $Pr(A)$ the probability. High coverage intervals can be used to produce normalized probability measures $att = (I_i, \frac{C_i}{C})$, or simply $att = (I_i, C_i)$, for these databases.

4.4 The stability score for query answers

The point statistics and high coverage intervals provide static information on how viable answers are distributed. However, most heterogeneous information systems, such as PDMSs, are dynamic and data sources may freely leave. One natural question is how to keep aggregate answer statistics up-to-date given that the departure of data sources will affect the aggregate answer distribution. To answer this question, we define the *stability* of the aggregate query.

Stability measures the amount of change caused in the viable answer distribution when some of the sources are removed. It can be quantified as the distance measure between the viable answer distribution without and with some sources removed. Given a number of data sources to be removed r, we randomly remove a set $\mathcal{Q}$ of size r from $\mathcal{D}$ and denote the resulting viable answer distribution by $f_X^{\mathcal{D}\setminus\mathcal{Q}}$. We use the distance measures introduced in Section 2.3 to quantify the difference between the two distributions. Let $\mathcal{S}_{uniS} = \{x_i\}$, be the sampled viable set, with regard to all the sources $\mathcal{D}$, and uniS sampling.

Definition 6. *[Stability score of an aggregation] Let $\mathcal{G}$ be the set of all possible choices for removing r sources from $\mathcal{D}$, and $Pr(\mathcal{Q})$ be the probability of choosing (a particular) $\mathcal{Q}$ with size r. Given $\mathcal{S}_{uniS}$, we define the stability score of the aggregation as*

$$
\begin{aligned}
Stab_d(\mathcal{S}_{uniS}) &\doteq -\log\left(\mathbb{E}[d(f_X^{\mathcal{D}}, f_X^{\mathcal{D}\setminus\mathcal{Q}})]\right)\\
&= -\log\left(\sum_{\mathcal{Q}\in\mathcal{G}} Pr(\mathcal{Q})d(f_X^{\mathcal{D}}, f_X^{\mathcal{D}\setminus\mathcal{Q}})\right)
\end{aligned}
\tag{4.1}
$$

where $\mathbb{E}$ denotes expectation and d is a distance measure. □

Note that $f_X^{\mathcal{D}}$ is a constant distribution; however, random removal of r sources results in random distributions $f_X^{\mathcal{D}\setminus\mathcal{Q}}$, and subsequently random values for $d(f_X^{\mathcal{D}}, f_X^{\mathcal{D}\setminus\mathcal{Q}})$. We use the expectation of the difference between the two distributions, $\mathbb{E}[d(f_X^{\mathcal{D}}, f_X^{\mathcal{D}\setminus\mathcal{Q}})]$, as the stability measure for the answer distribution. We use the negative logarithmic over the expected distribution difference as the stability score so that a higher value indicates a higher stability score, i.e., that the viable answer distribution is expected to change less against data source changes.

Furthermore, note that we do not need to enumerate or choose $\mathcal{Q}$, and explicitly compute $f_X^{\mathcal{D}\setminus\mathcal{Q}}$; this is just for analysis. The computation of the L_2^2 stability score does not rely on it (Section 2.3).

When additional knowledge regarding the likelihood of the departure of sources is not available, we apply an equal chance assumption on data source removals, where $\Pr(\mathcal{Q}) = 1/\binom{|\mathcal{D}|}{r}$ is constant given r. Another assumption needed for stability analysis is that the number of data sources to be removed is small i.e., $r \ll |\mathcal{D}|$. This assumption is viable, because when the number is large, the system will re-evaluate all the queries regardless of the stability scores. A key benefit to the stability analysis is that it helps prioritize which queries need updating when sources are updated with new values.

Now we describe how to compute the stability score. Given $\mathcal{S}_{uniS} = \{x_i\}$, the sampled viable set, the answer distribution is estimated by $f_X^{\mathcal{D}}(x) = \frac{1}{|\mathcal{S}_{uniS}|h}\sum_{i=1}^{|\mathcal{S}_{uniS}|} K(\frac{x-x_i}{h})$ (Section 4.2). Let $\mathcal{R}_{\mathcal{Q}}$ be the set of sample points that belong to the removed sources $\mathcal{Q}$. To estimate the new distribution, we need to exclude the points in $\mathcal{R}_{\mathcal{Q}}$

$$
\begin{aligned}
f_X^{\mathcal{D}\setminus\mathcal{Q}}(x) &= \frac{1}{(n-|\mathcal{R}_{\mathcal{Q}}|)h}\sum_{x_i\notin\mathcal{R}_{\mathcal{Q}}} K(\frac{x-x_i}{h})\\
&= \frac{n}{n-|\mathcal{R}_{\mathcal{Q}}|} f_X^{\mathcal{D}}(x) - \frac{1}{(n-|\mathcal{R}_{\mathcal{Q}}|)h}\sum_{x_i\in\mathcal{R}_{\mathcal{Q}}} K(\frac{x-x_i}{h})
\end{aligned}
\tag{4.2}
$$

where $n = |\mathcal{S}_{uniS}|$. The first term with $f_X^{\mathcal{D}}(x)$ is constant given a query, but the second term changes with different choices of $\mathcal{R}_{\mathcal{Q}}$.

We show in Theorem 4.2 that under the equal chance assumption for source removals, stability score can be obtained analytically for the L_2 distance measure.

Theorem 4.2 (L_2 stability score). *Given $\mathcal{S}_{uniS}$, its L_2^2 stability score*

$$Stab_{L_2}(\mathcal{S}_{uniS}) = -\frac{1}{2}\log\left(\frac{1}{2nh\sqrt{\pi}} * \frac{c_r}{1-c_r}(1-\frac{2}{n(n-1)}\Psi)\right) \tag{4.3}$$

where $n = |\mathcal{S}_{uniS}|$, h is the bandwidth parameter of the Gaussian kernel, the change ratio is estimated by $c_r = 1 -$

$(1 - \frac{y}{|\mathcal{D}|})^r$, *with y defined as the average number of sources needed for an answer, and the mutual impact factor is* $\Psi = \sum_{i,j} e^{-(x_i - x_j)^2/4h^2}$.

The proof to Theorem 4.2 is in Appendix A. This greatly reduces the computational overhead for the stability score. It eliminates the need of simulating source removal in order to compute a stability score.

We treat the change of viable answers due to data source removals as a random variable. The L_2 stability score is an estimator of the expectation of this random variable. We can also assess its 2nd moment, which gives the variance. To do this, we use the Bhattacharyya distance d_{Bh}, as the distribution difference measure and compute the stability score for the square of the viable answer distribution $\left(f_X^{\mathcal{D}}\right)^2$. Corollary 4.1 shows that this score can also be computed without source removal simulation.

Corollary 4.1. *Given $\mathcal{S}_{uniS}$, with n, h, and Ψ as defined in Theorem 4.2, the Bh (Bhattacharyya distance) stability score over the square of the viable answer distribution, is*

$$Stab_{Bh}(\mathcal{S}_{uniS}) = -\log\left(\frac{1}{2nh\sqrt{\pi}} + \frac{1}{n^2 h\sqrt{\pi}}\Psi\right) \quad (4.4)$$

Proving the corollary requires the same technique for Theorem 4.2 (in Appendix A). Since the expectation of the density $f_X^{\mathcal{D}\setminus\mathcal{Q}}$ is just $f_X^{\mathcal{D}}$, simple calculation by changing the order of expectation and summation, the result follows.

The stability score measures the likelihood of changes to query answers along with data source availability and updates. It can be used to prioritize the re-evaluation and update of queries, especially in a scenario where multiple continuous queries are managed. Note that the system needs to maintain neither the sampled viable answers nor the density estimation. A priority queue of the stability scores for the continuous queries is sufficient for maintenance.

5. EMPIRICAL STUDY

Dataset: We empirically tested the extraction of aggregate statistics using synthetic and real-life datasets. The synthetic tests allowed us to scale various parameters to verify the observations and predictions made in the analysis. For the real-life data, we experimented with Canadian climate data [8]. This archive contains official weather observations from stations located across Canada. The stations report hourly, daily, and monthly data measurements for attributes including, but not limited to: mean temperature, maximum temperature, total rainfall, total snow, direction of maximum gust, and total precipitation. Not all of the data is quality controlled. Furthermore, the sources may have missing values for some attributes, which typically implies the data had not been observed. In addition to the web interface, the data can be downloaded in XML or CSV format. We used the monthly climate data for the year 2006, from 1672 stations measuring climate data for 104 districts. Tables 1 and 2 summarize our data sets.

Aggregate Query: The query we use in all experiments sums temperature climate data over 500 components from the different data sets in Table 1 ($|\mathcal{C}| = 500$).

Settings: The algorithms were implemented in Matlab version 7.6.0 and the experiments were run on a PC with 2.5GHz Intel Core 2 duo CPU.

Data	Notes	Parameters
D2	A mixture of four Gaussians	$\mu \in [10,20], [25,35], [40,50], [55,65], \sigma = 0.5, weight = 12:5:2:1$
D3	A mixture of Gaussians, Cauchy and Gamma	$\mu \in [10,20]$, $\sigma = 1, \infty, 1$
C	(Real-life) Monthly climate data 2006	1672 stations, 104 measuring districts

Table 1: Data set details

Parameter	Symbol	Default
#Data sources	$\|\mathcal{D}\|$	100
Aggregation size	$\|\mathcal{C}\|$	500
uniS sample size	$\|\mathcal{S}_{uniS}\|$	400
#Bootstrap sample sets	$\|\mathcal{S}_{boot}\|$	50
Bootstrap sample size	$\|\mathcal{B}^i_{boot}\|$	$\|\mathcal{S}_{uniS}\|$
Confidence level	$1-\alpha$	90%

Table 2: Parameters in empirical study

5.1 Sampling and bootstrap improvements

Table 3 shows the improvements that are obtained by using the bootstrap method compared to direct inference. In particular, we report improvements in deriving tight confidence intervals for point statistics, and on savings of the required sample size. These experiments were conducted on dataset D2.

In Table 3, the confidence interval length from direct inference denoted by $len(CI_{di})$, is used as the baseline for each individual sampling. We report the maximal and average improvements using an improvement ratio defined as $i_r = \frac{len(CI_{di})}{len(CI_{boot})}$, where $len(CI_{boot})$ is the confidence interval length from bootstrapping. Smaller confidence intervals represent more reliable estimates; thus, greater values of i_r show bootstrapping achieves better performance. We can see that by using bootstrap sampling (with BC_a) with sample sizes of 200 and 400, the average improvement ratio is approximately 2, which shows confidence intervals returned by bootstrapping are half of that guaranteed by direct inference method. They are 3 to 4 times tighter when the sample size is small (200).

Table 3 also shows the savings on the required sample size in order to reach the confidence interval achieved by using the bootstrap method. Similar to the improvement ratio for Table 3, the tighter confidence intervals are translated to the savings on the required size of samples if the same confidence interval length that bootstrapping reports needs to be achieved with direct inference. Hence the saving ratio is defined as $s_r = \frac{|\mathcal{S}_{di}|}{|\mathcal{S}_{uniS}|}$ where $|\mathcal{S}_{di}|$ is the sample size that direct inference needs, and $|\mathcal{S}_{uniS}|$ is the initial sample size that bootstrapping uses. We can see from Table 3 that the average savings on the sample size is about a factor of 4. The results indicate that the confidence interval reported with bootstrap sampling is much tighter than using direct inference especially when the theoretical upper-bound of variance is large.

5.2 High coverage intervals

We present the results of high coverage intervals detected for 4 *sum* aggregations S_1 to S_4 where S_1, and S_2 sum climate data in C and S_3 and S_4 sum values generated in D_3. Figure 7 shows that the viable answer distributions for the 4 aggregates are multi-modal, and when components in

$\lvert S_{uniS}\rvert$	$1-\alpha$	max i_r	avg i_r	max s_r	avg s_r
200	0.8	4.248	2.556	18.1	7.36
200	0.9	3.309	2.119	10.96	4.84
400	0.8	2.896	2.001	8.39	4.28
400	0.9	2.293	1.655	5.26	2.82

Table 3: Bootstrapping improves confidence intervals and the savings on required sample size.

Fig	Greedy	Optimal	Cover	Greedy/Optimal
a	0.2272	0.2272	85.72%	1.0
b	0.2475	0.2475	85.44%	1.0
c	0.3764	0.2724	73.82%	1.38
d	0.5552	0.5150	92.12%	1.08

Table 4: Approximation ratio of the CIO algorithm

the aggregation are differently distributed, the viable answer distribution has different modes (7 modes in Figure 7.c compared to 8 in Figure 7.d).

By returning intervals in "dense" areas, the intervals cover a small percentage of the range of data (under 25% for S_1, S_2 with 2 modes, 37% for S_3 with 7 modes) to cover the majority of the distribution. The mean of all 4 is in the central flat area; expanding confidence intervals centering at the distribution's mean will result in very large confidence intervals.

Table 4 compares the performance of the greedy algorithm with an optimal method that slices the range of the density function uniformly into 4096 pieces and sums the top t slices that cover the desired probability measure. The greedy approximation is better if the "Greedy/Optimal" value is closer to 1.0 (it is always ≥ 1.0). Note that although the optimal method is more likely to return "tighter' intervals , it does not guarantee the continuity of the returned intervals; thus we used the greedy method in our solution.

5.3 Stability

We designed a simulation process to test the effectiveness and sensitivity of the L_2 stability score. We used the same aggregations as Section 5.2. The simulation worked as follows: we deleted one data source from the 100 sources and drew samples from viable answers that are computable from the remaining 99 sources (e.g., for climate data set, it is 1 out of 104 reporting districts). We recorded the means of the viable answers computed from 99 data sources (103 sources for climate data set). The *deviation maps* in Figure 8 correspond to the four distributions in Figure 7. They show the changes on the sample means when different data sources are disabled. For each circular graph, the center is the mean $\mu^{\mathcal{D}}$ of the viable distribution when no data source is removed; the points represent the means $\mu^{\mathcal{D}\setminus\mathcal{Q}}$ of the viable answer distribution when different data sources are removed. The distance between the data points and the center is defined as $d = \frac{|\mu^{\mathcal{D}\setminus\mathcal{Q}} - \mu^{\mathcal{D}}|}{\mu^{\mathcal{D}}}$.

Comparing the four deviation maps in Figure 8, we can see that answer distributions that have a higher stability score are more "stable", as demonstrated by the fact that the viable distribution means are more densely populated around the center. Note that the L_2 stability does not directly assess the change of distribution means. For example, Figure 2 suggests that two distributions with large differences may have the same mean. We observed this as a consistent trend in our empirical study. While we can confirm that queries with higher L_2 stability scores are more stable, we are not yet able to answer questions like "Will the mean of viable answers shift for more than 10% for a query with score 6.3?" Our suggested use of the stability score is for prioritizing updating of queries by re-evaluating queries with lower stability scores. We plan to focus our future work on deeper evaluation of similarity scores and stability scores when more than one source is removed.

5.4 Processing overhead of operations

Figure 6 reports the execution times of three of the main operations consisting of bootstrap re-sampling, KDE, and greedy CIO algorithm. The time for computing stability scores is negligible and has been discarded (200 iterations required less than a millisecond). Networking overhead times have also been ignored.

As indicated by Figure 6, KDE dominates the processing overhead of the operations. In the experiments, we use 50 bootstrap sample sets with different sizes. The bootstrapping time increases with the sample size but takes less than $3/50 = 60ms$ per run. KDE takes about 5 seconds on 50 sample sets of size 800. The greedy CIO algorithm running time is constant as the sample size increases, since a density function with constant number of 4096 points, is used. We estimate the time needed for computing one viable answer to be $200ms$, which is optimistic since sampling over a distributed hierarchy usually takes up to several seconds when the networking overhead is considered. Therefore, sampling the viable answers dominates the overall time needed for sampling and extracting statistics (e.g., 80 seconds in sampling and 5 seconds to extract statistics). This suggests that our technique is fast and further optimizations should focus on more efficient aggregate computation.

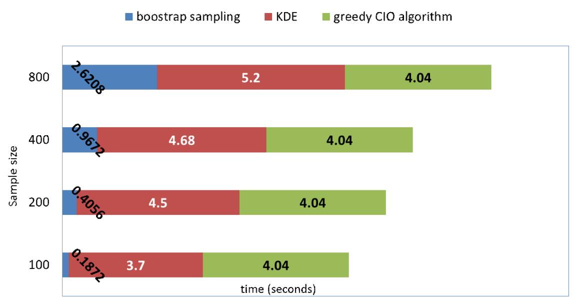
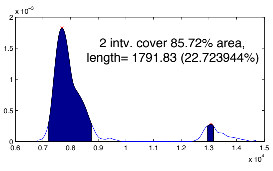


Figure 6: Time breakdown of operations.

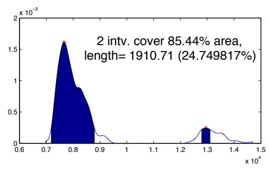
6. RELATED WORK

Data integration is concerned with increasing the coverage of data, and with representing it concisely and accurately [5, 11]. The first objective is typically realized by adding more sources. However, it is argued in [12] that "the more the better" is not always true for integration; obtaining, cleaning, and integrating data can be costly. Furthermore, adding low-quality sources can deteriorate integration quality. Instead, [12] proposes to balance integration cost and gain by selecting a subset of the sources wisely. Our sampling algorithm can be extended with similar ideas.

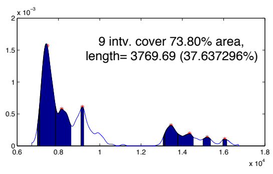
To realize the second objective, concise and accurate data representation, heterogeneity across data sources must be



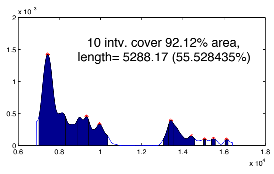
(a) S_1 (Climate Data C)



(b) S_2 (Climate Data C)

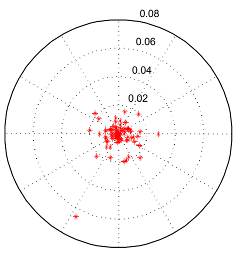


(c) S_3 (Mixture of 3 distributions - dataset D3)

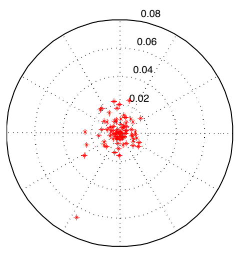


(d) S_4 (Mixture of 3 distributions - dataset D3)

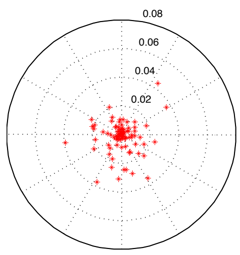
Figure 7: Multi-modal distributions and high coverage intervals



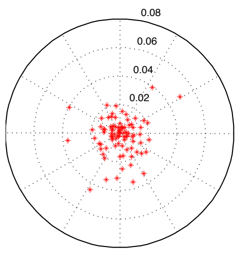
(a) S_1 L_2 score=6.5882



(b) S_2 L_2 score=6.4139



(c) S_3 L_2 score=6.4217



(d) S_4 L_2 score=6.3204

Figure 8: Deviations of empirical means when a single data source is disabled. Numbers indicate the relative distance ($0.02 = 2\%$) from the center. Higher stability scores correspond to figures that have denser distributions around the center.

resolved. As previously mentioned, [19] divides heterogeneity in heterogeneous information systems in three levels: schema (determining which schema elements correspond to each other), instance (determining which data instances correspond to each other), and value (given conflicting values for corresponding instances, choose which one to use). Our work focuses on the third objective. Data fusion [5, 11], also works on the third objective. It assumes a single true value for each component in a data set, and attempts to resolve value conflicts among the sources. In [18] information conflicts are resolved by estimating the reliability of sources and truth values in a joint inference on data with heterogeneous types. In our work, however, do not assume a single true value for components; instead we report a range of possible answers and aim to increase the users understanding and confidence in the reported results. In [19] the truthfulness of data on the Deep web, in particular, the flight and stock domain, is studied. In both domains, large amounts of redundancy and inconsistency at the data value level are observed. Furthermore, state of the art data fusion methods are also compared to resolve conflicts and estimate true values. Semantic ambiguity, out-of-date data, and pure errors are identified as reasons for inconsistent values across the sources. Semantic ambiguity, one the major reasons for value inconsistency, results from different semantics applied by the sources for the attributes they store. For example, one source may compute a statistic of the data over a year-long period, another may compute the same statistic over a half-year period. Both computations are correct with regard to the semantics applied; hence multiple true values are possible [19].

Substantial previous work on combining conflicting data values from multiple sources comes from wireless sensor network research [10, 15, 20, 21]. In a sensor network, sensor motes form an ad-hoc network and collaborate to transmit their sensor probe readings to a centralized repository that is usually beyond the range of a single mote. This is performed by transmitting data in a hierarchical aggregate network rooted at the central repository. Although the hierarchical aggregate network in our case has a lot in common with sensor networks, the operations are essentially different. The aggregate queries we process usually request a (small) part of data maintained in the distributed data sources; in the sensor network case, sensor motes have to upload all their data to the central repository. This results in different optimizations. We face the source combinatorial explosion and use sampling to make estimations, while a sensor network optimizes routing, transmission cost and seeks load balancing among battery powered sensor motes. In [23], the authors described a protocol to adaptively request data from remote sensors over time so as to control the error of approximate answers, while our work focuses on the snapshots of answer distributions when an aggregation is processed. The stability score helps maintain continuous queries but is not a direct approximation measure for a query.

In [14] a method to efficiently compute probability distribution functions in probabilistic databases, is proposed. While using such techniques could eliminate the need for sampling in many cases, in our scenario one of the goals is to reduce the need for data to be gathered from the distributed sources. Since we do not assume that we control the individual sources, we cannot ensure that they have the capabilities proposed in [14].

Research in databases with uncertainty, like our work, covers contexts that do not have one true answer. Various models exist, including uncertain databases [22], possibilistic [24] and probabilistic [16] databases, inconsistent databases [3]. Similarly, in data exchange, aggregate semantics with possible worlds [2] is discussed — there are many possible values that must be considered to find “the answer”.

Among the various models, the uncertain database [22] is closest to our setting: it also uses the relational model and the value of an attribute is a discrete distribution with positive probability on a finite set of values. This is similar to our setting, where the values from different data sources are transformed into values in a distribution. In [22] the authors discussed processing aggregate queries on uncertain databases where aggregates uses expectations of values in computation. Their work avoids the combinatorial explosion simply by disallowing exhaustive aggregates and does not give information about the viable answer distribution. In our semantic integration setting, computing the expectation of all component values is impractical as it requires collecting all single values from all data sources. Also, in our stability model, removal of one source will result in invalidating all components' values on that source. Therefore, these techniques cannot be applied to processing aggregates and providing distribution information.

7. CONCLUSION AND FUTURE WORK

This paper proposed our solution for estimating the distribution statistics for viable answers to aggregate queries in integration settings. Our technique extracts essential distribution statistics, returns intervals where aggregate answers have a high chance to be covered, and provides numerical stability scores for aggregate queries. We optimized the computation of the desired statistics using sampling and bootstrapping to minimize the sampling overhead and improved the confidence interval for point estimates of mean and variance. The greedy algorithm computes high coverage intervals quickly and provides good approximations. Our analysis enables the computation of stability scores without simulating source removal. All the optimizations allow the answer distribution estimation techniques described in this paper to be used with a query processing engine.

The stability analysis is the beginning of our investigation on monitoring query answers against changes in the data sources. We will try to establish links between the actual changes on aggregate answers and the stability score values as future work. In addition, the current uniS sampling algorithm assumes equal importance for the sources and samples them uniformly and independently. However, the sources may have different levels of quality and coverage. Future work should consider some notion of provenance. Furthermore, uniS is greedy. While this means that the sampling is not uniform on all viable answers, its speed may be an advantage. In addition, uniS can be fully parallelized as samples are obtained independently. Future work should examine how the algorithm scales when parallelized.

Another future direction, is to make inferences regarding the data and the sources based on the non-normality of the estimated viable answer distribution. In particular, the viable answer distribution can be used to diagnose possible errors in the data. Multi-modal distributions can indicate possible mapping problems in data integration. For example, the second high coverage interval in Figure 7 (a) is caused by combining supposedly cleaned data sets that incorrectly had values in both Fahrenheit and Celsius. Our work can be extended to help automatically detect such errors. Furthermore, using data stratification we can identify homogeneous data sources that apply similar semantics in their computations.

Acknowledgements

We thank Melsa Smith, Mohammad Ghasemi, and Omar AlOmeir for helpful discussions, and the anonymous reviewers for their constructive comments. This work has been funded by NSERC Canada and the NSERC BIN Strategic Network.

8. REFERENCES

[1] F. N. Afrati and R. Chirkova. Selecting and using views to compute aggregate queries (extended abstract). In *ICDT*, pages 383–397, 2005.

[2] F. N. Afrati and P. G. Kolaitis. Answering aggregate queries in data exchange. In *PODS*, 2008.

[3] M. Arenas, L. Bertossi, J. Chomicki, X. He, V. Raghavan, and J. Spinrad. Scalar aggregation in inconsistent databases. *Theoretical Computer Science*, 296:405–434, March 2003.

[4] A. Bhattacharyya. On a measure of divergence between two statistical populations defined by their probability distribution. *Bulletin of the Calcutta Mathematical Society*, 35:99–110, 1943.

[5] J. Bleiholder and F. Naumann. Data fusion. *ACM Comput. Surv.*, 41(1):1:1–1:41, Jan. 2009.

[6] Z. I. Botev, J. F. Grotowski, and D. P. Kroese. Kernel density estimation via diffusion. *Annual of Statistics*, 38(5), 2010.

[7] L. Breiman and L. Breiman. Bagging predictors. In *Machine Learning*, pages 123–140, 1996.

[8] Climate Canada. Canada climate data. `http://climate.weatheroffice.gc.ca/climateData/canada_e.html`, 2010.

[9] S. Cohen, W. Nutt, and A. Serebrenik. Rewriting aggregate queries using views. In *PODS*, pages 155–166, 1999.

[10] A. Deshpande, C. Guestrin, S. R. Madden, J. M. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. In *VLDB*, pages 588–599, 2004.

[11] X. L. Dong and F. Naumann. Data fusion: resolving data conflicts for integration. In *VLDB*, pages 1654–1655, 2009.

[12] X. L. Dong, B. Saha, and D. Srivastava. Less is more: selecting sources wisely for integration. In *VLDB*, pages 37–48, 2013.

[13] B. Efron. Better bootstrap confidence intervals. *Journal of the American Statistical Association*, 82(397):pp. 171–185, 1987.

[14] R. Fink, L. Han, and D. Olteanu. Aggregation in probabilistic databases via knowledge compilation. In *VLDB*, pages 490–501, 2012.

[15] N. Jain, D. Kit, P. Mahajan, P. Yalagandula, M. Dahlin, and Y. Zhang. Star: Self-tuning aggregation for scalable monitoring. In *VLDB*, 2007.

[16] T. S. Jayram, S. Kale, and E. Vee. Efficient aggregation algorithms for probabilistic data. In *SODA*, 2007.

[17] Joint infrastructure interdependencies research program—public safety canada. `http://www.civil.engineering.utoronto.ca/staff/professors/eldiraby/News/JIIRP_Project.htm`.

[18] Q. Li, Y. Li, J. Gao, B. Zhao, W. Fan, and J. Han. Resolving conflicts in heterogeneous data by truth discovery and source reliability estimation. In *SIGMOD*, pages 1187–1198, 2014.

[19] X. Li, X. L. Dong, K. Lyons, W. Meng, and D. Srivastava. Truth finding on the deep web: is the problem solved? In *VLDB*, pages 97–108, 2013.

[20] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. In *OSDI*, 2002.

[21] S. Madden, R. Szewczyk, M. Franklin, and D. Culler. Supporting aggregate queries over ad-hoc wireless sensor networks. In *Workshop on Mobile Computing and Systems Applications*, pages 49–58, 2002.

[22] R. Murthy and J. Widom. Making aggregation work in uncertain and probabilistic databases. *TKDE*, 2010.

[23] C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. In *SIGMOD*, pages 563–574, 2003.

[24] E. Rundensteiner and L. Bic. Evaluating aggregates in possibilistic relational databases. *DKE*, 7:239–267, 1992.

[25] J. Xu and R. Pottinger. Integrating domain heterogeneous data sources using decomposition aggregation queries. *Information Systems*, 39(0):80 – 107, 2014.

APPENDIX

A. PROOF OF THEOREM 4.2

Proving Theorem 4.2 requires first some background discussion on the product of two Gaussians (Section A.1) and then computing the integral of the square of the density estimation function (Section A.2)

A.1 The product of two Gaussians

Let $f_1(x) \sim \mathcal{N}(\mu_1, \sigma^2)$ and $f_2(x) \sim \mathcal{N}(\mu_2, \sigma^2)$:

$$\begin{aligned} f_1(x)f_2(x) &= \left(\frac{1}{\sqrt{2\pi\sigma^2}}\right)^2 e^{-\frac{(x-\mu_1)^2+(x-\mu_2)^2}{2\sigma^2}} \\ &= \frac{1}{2\sqrt{\pi\sigma^2}}\mathcal{N}(\frac{\mu_1+\mu_2}{2}, \frac{\sigma^2}{2})e^{-\frac{(\mu_1-\mu_2)^2}{4\sigma^2}} \quad \text{(A.1)} \end{aligned}$$

As a special case, let $f(x) = \frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{(x-\mu)^2}{2\sigma^2}} \sim \mathcal{N}(\mu, \sigma^2)$:

$$\begin{aligned} f^2(x) &= \frac{1}{\sqrt{2\pi\sigma^2}}\left[\frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{(x-\mu)^2}{\sigma^2}}\right] \\ &= \frac{1}{2\sqrt{\pi\sigma^2}}\mathcal{N}(\mu, \frac{\sigma^2}{2}) \quad \text{(A.2)} \end{aligned}$$

A.2 The integral of the square of the density function:

With the preparation in Appendix A.1, we can compute the summation of the square of the density estimation function. Recall in KDE using Gaussian kernel, the density function is estimated as $f(x) = \frac{1}{nh}\sum_1^n K(\frac{x-x_i}{h})$, where n is the size of the sample set. We now expand the integral on the square of the density, let $\alpha = \int f^2(x)dx$:

$$\begin{aligned} \alpha &= \int \frac{1}{(nh)^2}\left[\sum_{i=1}^n K(\frac{x-x_i}{h})\right]^2 dx \quad \text{(A.3)} \\ &= \frac{1}{(nh)^2}\int \sum_{i=1}^n K^2(\frac{x-x_i}{h})dx \quad \text{(A.4)} \\ &+ \frac{2}{(nh)^2}\int \sum_{i,j} K(\frac{x-x_i}{h})K(\frac{x-x_j}{h})dx \quad \text{(A.5)} \end{aligned}$$

The kernel function is Gaussian $K(\frac{x-x_i}{h}) = \frac{1}{\sqrt{2\pi}}e^{-\frac{(x-x_i)^2}{2h^2}}$, so $\frac{1}{h}K(\frac{x-x_i}{h}) \sim \mathcal{N}(x_i, h^2)$.

Switching the integral and summation in Equation (A.4), it is is simplified to

$$\frac{1}{2nh\sqrt{\pi}}$$

Similarly, Equation (A.5) is simplified to

$$\frac{1}{n^2h\sqrt{\pi}}\sum_{i,j} e^{-\frac{(x_i-x_j)^2}{4h^2}}$$

We can see that the integral (α) is a function over all the data points.

Now recall the stability analysis using the square d_{L_2} distance. Let $f_X^{\mathcal{D}}(x)$, $f_X^{\mathcal{D}\setminus\mathcal{Q}}(x)$ be the original and augmented density. We want to compute $\mathbb{E}\left[\int\left(f_X^{\mathcal{D}\setminus\mathcal{Q}}(x) - f_X^{\mathcal{D}}(x)\right)^2 dx\right]$. Since $\mathbb{E}[f_X^{\mathcal{D}\setminus\mathcal{Q}}(x)] = f_X^{\mathcal{D}}(x)$, this is the integral of the point-wise variance of the augmented function. It thus equals

$$d_{L_2^2} = \mathbb{E}\left[\int\left(f_X^{\mathcal{D}\setminus\mathcal{Q}}(x)\right)^2 dx\right] - \int\left(f_X^{\mathcal{D}}(x)\right)^2 dx$$

Using the above results on Gaussian square, and letting

$$\Psi = \sum_1^n e^{-(x_i-x_j)^2/4h^2} \quad \text{(A.6)}$$

$$f_X^{\mathcal{D}\setminus\mathcal{Q}} = \frac{1}{h(n-\mathcal{R}_\mathcal{Q})}\sum_1^{n-\mathcal{R}_\mathcal{Q}} e^{-(x_i-x_j)^2/4h^2} \quad \text{(A.7)}$$

$$\mathbb{E}[\mathcal{R}_\mathcal{Q}] = c_r * n \;\; for\, some\, constant\, c_r \quad \text{(A.8)}$$

we have

$$d_{L_2^2} = \frac{1}{2nh\sqrt{\pi}} * \frac{c_r}{1-c_r}(1 - \frac{2}{n(n-1)}\Psi) \quad \text{(A.9)}$$

and thus the stability score formula in Theorem 4.2 follows.

We can see that the distance is related to the viable answer distribution, $f_X^{\mathcal{D}}$, and the average fraction of affected answers when some sources are removed. Also it is easy to verify that when all the data points coincide, the distance is 0, i.e., most stable (the corresponding stability score is ∞).

Now we assess how many answers are likely to be affected when r sources are removed from a total of $|\mathcal{D}|$ data sources. This number relates to the number of data sources required for an answer. Suppose on average we need y (also called the weight) out of $|\mathcal{D}|$ data sources for a viable answer; then an estimate for the fraction that got affected is $c_r = \frac{\binom{|\mathcal{D}|}{y} - \binom{|\mathcal{D}|-r}{y}}{\binom{|\mathcal{D}|}{y}}$. We acknowledge that not all y combinations are answers to the query but this is still a good estimate when information regarding the coverage of the sources is not available. Moreover, the weight itself includes some of this information. The larger the average source coverage for components, the smaller the value of y.

Another way is to estimate the expected number of samples that become invalid when r sources are randomly removed This can be done by simulation with the sample set or using $c_r = 1 - (1 - \frac{y}{|\mathcal{D}|})^r$ which assumes that data sources uniformly contribute to aggregate answers. This completes the proof of Theorem 4.2.

In the proof we use the property of Gaussian distributions which limits the choice of kernel in KDE to Gaussian kernels. We note here that from the perspective of convergence, when the sample sizes increases, using any kernel will converge to the true distribution; thus the expectation of the changes computed here should also converge. Therefore if using any other kernels would make any difference, the divergence is from KDE, but it does not impact the stability analysis.

Discovering Recurring Patterns in Time Series

R. Uday Kiran[†], Haichuan Shang[†], Masashi Toyoda[†] and Masaru Kitsuregawa[†‡]

[†]The University of Tokyo, Tokyo, Japan
[‡]National Institute of Informatics, Tokyo, Japan
{uday_rage, shang, toyoda, kitsure}@tkl.iis.u-tokyo.ac.jp

ABSTRACT

Partial periodic patterns are an important class of regularities that exist in a time series. A key property of these patterns is that they can start, stop, and restart anywhere within a series. We classify partial periodic patterns into two types: (i) regular patterns − patterns exhibiting periodic behavior throughout a series with some exceptions and (ii) recurring patterns − patterns exhibiting periodic behavior only for particular time intervals within a series. Past studies on partial periodic search have been primarily focused on finding regular patterns. One cannot ignore the knowledge pertaining to recurring patterns. This is because they provide useful information pertaining to seasonal or temporal associations between events. Finding recurring patterns is a non-trivial task because of two main reasons. (i) Each recurring pattern is associated with temporal information pertaining to its durations of periodic appearances in a series. Obtaining this information is challenging because the information can vary within and across patterns. (ii) Finding all recurring patterns is a computationally expensive process since they do not satisfy the anti-monotonic property. In this paper, we propose recurring pattern model by addressing the above issues. We also propose Recurring Pattern growth algorithm along with an efficient pruning technique to discover these patterns. Experimental results show that recurring patterns can be useful and that our algorithm is efficient.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications - Data Mining.

General Terms

Algorithms

Keywords

Data mining, periodic pattern mining, time series

1. INTRODUCTION

A time series is a collection of events obtained from sequential measurements over time. Periodic pattern mining involves finding all patterns that exhibit either complete or partial cyclic repetitions in a time series. Past studies on partial periodic search have been focused on finding *regular patterns*, i.e., patterns exhibiting either complete or partial cyclic repetitions throughout a series [1, 2, 3, 4, 5, 6, 7, 8, 9]. An example regular pattern of $\{Bat, Ball\}$ states that customers have been purchasing items 'Bat' and '$Ball$' almost every day throughout the year. A useful related type of partial periodic pattern is *recurring patterns*, i.e., patterns exhibiting cyclic repetitions only for particular time intervals within a series. An example recurring pattern of $\{Jackets, Gloves\}$ states that customers have often purchased '$Jackets$' and '$Gloves$' from 10-October-2012 to 26-February-2013 and from 2-November-2013 to 2-March-2014. The purpose of this paper is to discover recurring patterns by addressing mining challenges.

Recurring patterns are ubiquitous in very large datasets. In many real-world applications, they can provide useful information pertaining to seasonal or temporal associations between items. In retail, a user may be interested in determining seasonal purchases for efficient inventory management. Similarly, a social network data analyst may be interested in obtaining temporal information pertaining to bursts of hashtags, such as #earthquakes, #radiation and #floods. Also, an expert in the health-care sector may be interested in finding seasonal diseases in a geographical location. To improve web site design and administration, an administrator may be interested in obtaining temporal information of heavily visited web pages. In the stock market, the set of high stocks indices that rise periodically for a particular time interval may be of special interest to companies and individuals. In a computer network, an administrator may be interested in finding high severity events (e.g. cascading failure) against regular routine events (e.g. data backup).

Unfortunately, finding recurring patterns is a non-trivial task because of the following reasons.

1. Each recurring pattern is associated with temporal information pertaining to its durations of periodic appearances within the data. Obtaining this information is challenging because the information can vary within and across patterns.

2. Most current periodic pattern mining approaches take into account a time series as a symbolic sequence; therefore, they do not take into account the actual temporal

10.5441/002/edbt.2015.10

information of events.

3. Recurring patterns do not satisfy the *anti-monotonic property.* That is, all non-empty subsets of a recurring pattern may not be recurring patterns. This increases the search space, which in turn increases the computational cost of finding these patterns. Therefore, developing efficient pruning techniques to reduce the search space is challenging.

4. Since regular patterns exhibit periodic behavior throughout a series with some exceptions, regular pattern mining algorithms do not obtain temporal information pertaining to the durations of periodic appearances of a pattern within the series. As a result, these algorithms cannot be extended for finding recurring patterns.

5. In real-life, recurring patterns involving rare items can be interesting to users. For example, the knowledge pertaining to rare events, such as cascading failures, are more important than regular events for a network administrator. However, finding such patterns is difficult since rare items appear infrequently in the data. Classifying items into frequent or rare is subjective and depends on the user and/or application requirements.

In this paper, we propose a model that addresses all the above-mentioned issues while finding recurring patterns. In particular, our model takes into account time series as a time-based sequence and models it as a transactional database with transactions ordered in respect to a particular timestamp (without loss of generality). Our model consists of three novel measures, *periodic-support*, *periodic-interval* and *recurrence*, to determine the dynamic periodic behavior of recurring patterns. *Periodic-support* determines the number of consecutive cyclic repetitions of a pattern in a subset of data. *Periodic-interval* determines the time interval (or window) pertaining to the periodic appearances of a pattern within a series. *Recurrence* determines the number of interesting *periodic intervals* of a pattern. Finally, we propose a pattern-growth algorithm along with an efficient pruning technique to discover recurring patterns effectively. We call our algorithm recurring pattern-growth (RP-growth). Experimental results show that RP-growth is efficient and recurring patterns can provide useful information in many real-life applications.

The rest of the paper is organized as follows. Section 2 describes related work on mining periodic patterns. Section 3 introduces our model of recurring patterns. Section 4 presents RP-growth. Sections 5 reports on the experimental results. Finally, Section 6 concludes the paper with future research directions.

2. RELATED WORK

Since the introduction of partial periodic patterns [5], the problem of finding these patterns has received a great deal of attention [6, 8, 10, 11, 12]. The model used in all these studies, however, remains the same. That is, it takes into account a time series as a symbolic sequence and finds all patterns using the following two steps:

1. Partition the symbolic sequence into distinct subsets (or period-segments) of a fixed length (or *period*).

2. Discover all partial periodic patterns that satisfy the user-defined *minimum support* (*minSup*), which controls the minimum number of period-segments that a pattern must cover though the sequence.

A major limitation of the above studies is that they do not take into account the actual temporal information of the events within a sequence. To address this issue, Ma and Hellerstein [7] modeled a time series as a time-based sequence and proposed a model to discover a class of partial periodic patterns known as **p-patterns**. In this model, a pattern is considered partial periodic if its number of periodic appearances throughout the sequence satisfies the user-defined *minSup*. It should be noted that the concept of *minSup* is not the same in both frequent pattern mining and partial periodic pattern mining. In frequent pattern mining, *minSup* controls the minimum number of appearances of a pattern throughout the data. However, in partial periodic pattern mining, *minSup* controls the minimum number of periodic appearances (or cyclic repetitions) of a pattern throughout the data. Thus, the partial periodic patterns discovered in all the above studies [6, 8, 10, 11, 12, 7] represent *regular patterns.* Our study, on the other hand, was focused on discovering recurring patterns in a time-based sequence. Moreover, Ma and Hellerstein's model cannot be extended for finding recurring patterns. The reasons are as follows:

1. Their model fails to obtain the temporal information pertaining to the durations of periodic appearances of a pattern within the data.

2. Finding p-patterns with a single *minSup* leads to the dilemma known as the "rare item problem" [13]. If *minSup* is set too high, those patterns that involve rare items will not be found. To find patterns involving both frequent and rare items, *minSup* has to be set very low. However, this can lead to combinatorial explosion producing too many patterns. In particular, many uninteresting aperiodic patterns can be discovered as partial periodic patterns. For example, if we set a low *minSup*, say *minSup* = 5%, then we will be discovering an uninteresting aperiodic pattern that has only 5% of its periodically appearances throughout the data as a partial periodic pattern.

Recently, researchers have been investigating the **complete cyclic behavior** of the frequent patterns in a transactional database to discover a class of user-interest-based patterns known as periodic-frequent patterns [9, 14, 15, 16]. Informally, a frequent pattern satisfying *minSup* is said to be **periodic-frequent** if and only if all its inter-arrival times throughout the database satisfy the user-defined *period* threshold value. Thus, these studies were focused on finding regular patterns in a transactional database. For our study, we investigated the partial cyclic behavior of the patterns to discover recurring patterns; thus, generalizing the current model of periodic-frequent patterns. More importantly, none of the approaches presented in [9, 14, 15, 16] model time series data as a transactional database. Instead, they are based on the implicit assumption that there are transactional databases with a sequentially ordered set of transactions. This paper fills the gap by describing the procedure to model time series data as a transactional database without loss of generality.

Yang et al. [17] investigated the change in periodic behavior of patterns due to the intervention of random noise and introduced a class of user-interest-based patterns known as *asynchronous periodic patterns*. Although asynchronous-periodic pattern mining is closely related to our work, it cannot be extended for finding recurring patterns. The reason is asynchronous periodic pattern mining models a time series as a symbolic sequence; therefore, it does not take in account the actual temporal information of the events within a sequence.

The problem of finding sequential patterns [18] and frequent episodes [19, 20] has received a great deal of attention. However, it should be noted that *periodicity* is not considered in these studies. Ozden et al. [2] investigated the problem of finding cyclic association rules. However, that study is quite restrictive in finding the patterns that are present at every cycle.

Finding partial periodic patterns [4], motifs [21], and recurring patterns [22] has also been studied in time series; however, the focus was on finding numerical curve patterns rather than symbolic patterns.

Overall, the proposed model of finding recurring patterns is novel and is distinct from current models. In the next section, we introduce our model of recurring patterns.

3. PROPOSED MODEL

In this section, we first describe time series as defined in [7]. Next, we represent these series as a transactional database and introduce measures to find recurring patterns.

Definition 1. *Let I be a set of items (or event types).* ***An event*** *is a pair (i, ts), where $i \in I$ is an item and $ts \in R$ is the timestamp of the event. Let $X \subseteq I$ be a* ***pattern***. ***An event sequence*** *S is an ordered collection of events, i.e., $\{(i_1, ts_1), (i_2, ts_2), \cdots, (i_N, ts_N)\}$, where $i_j \in I$ is an item at the j-th event. The term ts_j represents the occurrence timestamp of the event, and $ts_h \leq ts_j$ for $1 \leq h \leq j \leq N$ [7].*

Definition 2. *A* ***point sequence*** *is an ordered collection of occurrence times. Given an event sequence $S = \{(i_1, ts_1), (i_2, ts_2), \cdots, (i_N, ts_N)\}$, there is an implied point sequence, $\widehat{S} = \{ts_1, ts_2, \cdots, ts_N\}$. An event sequence can be viewed as a mixture of multiple point sequences of each item. Let TSD denote the time series data (or a set of events) being mined.*

Example 1. *Figure 1 shows a TSD with a set of items $I = \{a, b, c, d, e, f, g\}$. In this figure, an item of each event is labeled above its occurrence timestamp. It should be noted that no item appears at the timestamps of 8 and 13. The item 'a' appears at the timestamps of 1, 2, 3, 4, 7, 11, 12 and 14. Therefore, the event sequence of 'a' is represented as $S^a = \{(a, 1), (a, 2), (a, 3), (a, 4), (a, 7), (a, 11), (a, 12), (a, 14)\}$. The point sequence of 'a' is represented as $\widehat{S^a} = \{1, 2, 3, 4, 7, 11, 12, 14\}$. Similarly, the point sequences of 'b' and 'ab' are represented as $\widehat{S^b} = \widehat{S^{ab}} = \{1, 3, 4, 7, 11, 12, 14\}$.*

The point sequence plays an important role in assessing the periodic behavior of the patterns in a time series. We now describe the temporally ordered transactional database which preserves the point sequence of items in the TSD.

A **transaction**, $tr = (ts, Y)$, is a tuple, where ts represents the timestamp and Y is a pattern. A **transactional**

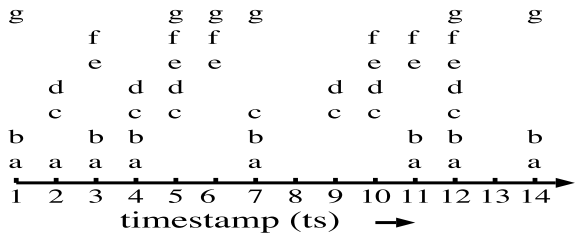


Figure 1: Running example: time-based sequence consisting of items from 'a' to 'g'

database TDB over I is a set of transactions, $TDB = \{tr_1, \cdots, tr_m\}$, $m = |TDB|$, where $|TDB|$ is the size of the TDB in total number of transactions. For a transaction $tr = (ts, Y)$, such that $X \subseteq Y$, it is said that X occurs in tr and such a timestamp is denoted as ts^X. Let $TS^X = \{ts_k^X, \cdots, ts_l^X\}$, where $1 \leq k \leq l \leq m$, denote an **ordered set of timestamps** at which X has occurred in the TDB. The TS^X in the TDB is the same as the point sequence of X in the TSD. Therefore, we do not miss any information pertaining to the temporal appearances of a pattern in the data.

Example 2. *Table 1 shows the transactional database constructed by grouping the items appearing together at a particular timestamp in Figure 1. Each transaction in this database is uniquely identifiable with a timestamp. All transactions have been ordered with respect to their timestamps. It can be observed that the constructed database does not contain the transactions with timestamps 8 and 13. The reason is that no item appears at these timestamps in Figure 1. In this database, the pattern 'ab' appears at the timestamps of* 1, 3, 4, 7, 11, 12, *and* 14. *Therefore, $TS^{ab} = \{1, 3, 4, 7, 11, 12, 14\}$.*

Table 1: Transactional database constructed from time-based sequence shown in Figure 1. The term 'ts' is an acronym for timestamp

ts	Items
1	*a, b, g*
2	*a, c, d*
3	*a, b, e, f*
4	*a, b, c, d*
5	*c, d, e, f,g*
6	*e, f, g*
7	*a, b, c, g*
9	*c, d*
10	*c, d, e ,f*
11	*a, b, e, f*
12	*a, b, c, d, e, f, g*
14	*a, b, g*

Definition 3. ***(Support of pattern X)*** *The number of transactions containing X in the TDB is defined as the support of X and denoted as $Sup(X)$. That is, $Sup(X) = |TS^X|$.*

Example 3. *The support of 'ab' in Table 1 is the size of TS^{ab}. Therefore, $Sup(ab) = |\{1, 3, 4, 7, 11, 12, 14\}| = 7$.*

Definition 4. ***(Periodic appearance of pattern X)*** *Let $ts_j^X, ts_k^X \in TS^X$, $1 \leq j < k \leq m$, denote any two consecutive timestamps in TS^X. The time difference between ts_k^X and ts_j^X is referred to as* ***an inter-arrival time*** *of X, and denoted as iat^X. That is, $iat^X = ts_k^X - ts_j^X$. Let $IAT^X = \{iat_1^X, iat_2^X, \cdots, iat_k^X\}$, $k = Sup(X) - 1$, be the* *set of all inter-arrival times of X in TDB. An inter-arrival time of pattern X is said to be* ***periodic*** *(or interesting) if it is no more than the user-defined period threshold value. That is, a $iat_i^X \in IAT^X$ is said to be* ***periodic*** *if $iat_i^X \leq per$, where 'per' represents the period.*

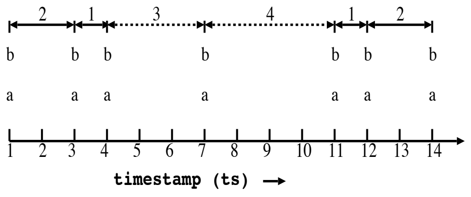


Figure 2: Inter-arrival times of '*ab*', IAT^{ab}

Example 4. *The pattern 'ab' has initially appeared at the timestamps of 1 and 3. The difference between these two timestamps gives an inter-arrival time of 'ab.' That is, $iat_1^{ab} = 2$ $(= 3 - 1)$. Similarly, other inter-arrival times of 'ab' are $iat_2^{ab} = 1$, $iat_3^{ab} = 3$, $iat_4^{ab} = 4$, $iat_5^{ab} = 1$ and $iat_6^{ab} = 2$. Therefore, the resultant $IAT^{ab} = \{2, 1, 3, 4, 1, 2\}$. If the user-defined $per = 2$, then iat_1^{ab}, iat_2^{ab}, iat_5^{ab} and iat_6^{ab} are considered the periodic occurrences of 'ab' in the data. On the other hand, iat_3^{ab} and iat_4^{ab} are considered the aperiodic occurrences of 'ab' as $iat_3^{ab} \not\leq per$ and $iat_4^{ab} \not\leq per$. Figure 2 shows the set of all inter-arrival times for pattern 'ab', i.e., IAT^{ab}. The thick lines represent the inter-arrival times that satisfy the period, while the dotted lines represent the inter-arrival times that fail to satisfy the period.*

Most current partial periodic pattern mining approaches use *minSup* to assess the periodic interestingness of a pattern [5]. This measure cannot be used for finding recurring patterns because it controls the minimum number of cyclic repetitions a pattern must have in all the data. Therefore, we introduce the following measures to determine the partial periodic behavior of recurring patterns.

Definition 5. ***(Periodic-interval of pattern X)*** *Let $TS_j^X = \{ts_p^X, \cdots, ts_q^X\} \subseteq TS^X$, $p \leq q$, be a set of timestamps such that $\forall ts_k^X \in TS_j^X$, $p \leq k < q$, $ts_{k+1}^X - ts_k^X \leq per$. The TS_j^X is a* ***maximal set*** *if there exists no superset in which an inter-arrival time between the two consecutive timestamps is no more than the period. The range of timestamps in TS_j^X represents a periodic-interval of X and is denoted as pi_j^X. That is, $pi_j^X = [ts_p^X, ts_q^X]$.*

Example 5. *The maximal sets of timestamps in which 'ab' has appeared within the user-defined $per = 2$ are: $TS_1^{ab} = \{1, 3, 4\}$, $TS_2^{ab} = \{7\}$, and $TS_3^{ab} = \{11, 12, 14\}$. Therefore, the corresponding periodic-intervals for 'ab' are $pi_1^{ab} = [1, 4]$, $pi_2^{ab} = [7, 7]$, and $pi_3^{ab} = [11, 14]$.*

The *periodic-interval*, as defined above, obtains information pertaining to the duration (or window) of periodic appearances of a pattern in a database. Most importantly, it can effectively determine the periodic durations that can vary within and across patterns. In very large databases, a pattern may have too many periodic-intervals. An efficient technique to reduce this number is to select only those periodic-intervals in which the number of cyclic repetitions of the corresponding pattern satisfies the user-defined threshold value. Thus, we introduce the following two definitions.

Definition 6. ***(Periodic-support of pattern X)*** *The size of TS_j^X is defined as the* periodic-support *of X, and denoted as ps_j^X. That is, $ps_j^X = |TS_j^X|$.*

Example 6. *The periodic-support of 'ab' in pi_1^{ab} is the size of $|TS_1^{ab}|$. Therefore, $ps_1^{ab} = |TS_1^{ab}| = 3$. Similarly, the periodic-supports of 'ab' in pi_2^{ab} and pi_3^{ab} are 1 and 3, respectively.*

In the real-world applications, some items appear very frequently in the data, while others rarely appear. We have observed that some rare items also exhibit periodic behavior in a portion of the data. The *periodic-support*, as defined above, facilitates the user to discover the knowledge pertaining to those frequent and rare items that have exhibited sufficient number of cyclic repetitions in a portion of database. Each *periodic-interval* of a pattern will have only one *periodic-support* and vice-versa. In other words, there is one-to-one relationship between the *periodic-intervals* and *periodic-supports* of a pattern.

Definition 7. ***(Interesting periodic-interval of pattern X)*** *Let $PI^X = \{pi_1^X, \cdots, pi_k^X\}$ and $PS^X = \{ps_1^X, \cdots, ps_k^X\}$, $1 \leq k$, be the complete set of periodic-intervals and periodic-supports of pattern X in the TDB, respectively. A $pi_k^X \in PI^X$ is said to be an interesting periodic-interval if its corresponding $ps_k^X \in PS^X$ has $ps_k^X \geq minPS$. The $minPS$ represents the user-defined minimum periodic-support.*

Example 7. *If the user-defined $minPS = 3$, then pi_1^{ab} and pi_3^{ab} are considered the interesting periodic-intervals of 'ab'. This is because $ps_1^{ab} \geq minPS$ and $ps_3^{ab} \geq minPS$. The pi_2^{ab} is considered an uninteresting periodic-interval of 'ab' as $ps_2^{ab} \not\geq minPS$.*

Since very large databases are generally composed over a very long time frame, it has been observed that some users may specify a constraint on the minimum number of interesting periodic-intervals. Thus, we introduce the following definitions.

Definition 8. ***(Recurrence of pattern X)*** *The* recurrence count *of a pattern represents its number of interesting periodic-intervals in a database. Let $IPI^X \subseteq PI^X$ be the set of periodic-intervals of X such that for every $pi_k^X \in IPI^X$, its corresponding $ps_k^X \geq minPS$. The* recurrence *of pattern X is denoted as $Rec(X) = |IPI^X|$.*

Example 8. *Continuing with the previous example, $IPI^{ab} = \{[1, 4], [11, 14]\}$. The recurrence of 'ab' is the size of IPI^{ab}. That is, $Rec(ab) = |IPI^{ab}| = 2$.*

Definition 9. ***(Recurring pattern X)*** *Pattern X is a* recurring pattern *if $Rec(X) \geq minRec$, where $minRec$ is the user-specified* minimum recurrence count. *Recurring pattern X is expressed as follows:*

$$X \quad [Sup(X), Rec(X), \{\{pi_k^X : ps_k^X\} | \forall pi_k^X \in IPI^X\}]. \quad (1)$$

Example 9. *If the user-defined $minRec = 2$, then 'ab' is a recurring pattern and is expressed as follows:*

$$ab \quad [support = 7, recurrence = 2, \{\{[1, 4] : 3\}, \{[11, 14] : 3\}\}].$$

The above pattern informs that 'ab' has occurred in 7 *transactions and its periodic occurrence behavior of once in every two transactions consecutively for at least three times has been observed at two distinct subsets of a database whose timestamps are in the range [1, 4] and [11, 14]. Table 2 shows the complete set of recurring patterns discovered from Table 1.*

Table 2: Recurring patterns in Table 1. Terms '*Sup*,' '*Rec*' and '*IPI*' respectively denote *support*, *recurrence*, and interesting periodic-intervals along with their periodic-supports

Pattern	*Sup*	*Rec*	*IPI*
a	8	2	{{[1,4]:4}, {[11,14]:3}}
b	7	2	{{[1,4]:3}, {[11,14]:3}}
d	6	2	{{[2,5]:3}, {[9,12]:3}}
e	6	2	{{[3,6]:3}, {[10,12]:3}}
f	6	2	{{[3,6]:3}, {[10,12]:3}}
ab	7	2	{{[1,4]:3}, {[11,14]:3}}
cd	6	2	{{[2,5]:3}, {[9,12]:3}}
ef	6	2	{{[3,6]:3}, {[10,12]:3}}

DEFINITION 10. ***(Problem Definition:)*** *Given a time-based sequence (i.e., a TSD), the problem of finding recurring patterns involve discovering all those patterns that satisfy the user-defined per, minPS and minRec constraints.*

The measures, *support*, *period* and *periodic-support*, can also be expressed in percentage of $|TDB|$. However, we use the former definitions for ease of explanation. Table 3 lists the nomenclature of different terms used in our model.

Table 3: Nomenclature of various terms used in our model

Terminology	Notation
The timestamp of a transaction containing X	ts_i^X
The set of all timestamps containing X	TS^X
The *support* of X	$Sup(X)$
An inter-arrival time of X	iat_i^X
The set of all inter-arrival times of X	IAT^X
The user-defined *period*	per
A *periodic-support* of X	ps_i^X
The set of all *periodic-supports* of X	PS^X
A *periodic-interval* of X	pi_i^X
The set of all *periodic-intervals* of X	PI^X
The set of interesting periodic-intervals of X	IPI^X
The *recurrence* of X	$Rec(X)$

The construction of a transactional database from a time series involves grouping the items appearing together at a particular timestamp and storing them in a linked hash table. As this process is simple and straight forward, we do not discuss it in this paper. Instead, we focus on finding the recurring patterns from the constructed database.

4. PROPOSED ALGORITHM

In this section, we first introduce our pruning technique to reduce the computational cost of finding recurring patterns. Next, we present our algorithm to mine the complete set of recurring patterns from the constructed database.

4.1 Basic Idea: Candidate patterns

The space of items in a database gives rise to a subset lattice. An itemset lattice is a conceptualization of search space while finding user-interest-based patterns. The **anti-monotonic property** has been widely used to reduce the search space [23]. Unfortunately, recurring patterns do not satisfy this property. This increases the search space, which in turn increases the computational cost of mining recurring patterns.

EXAMPLE 10. *Consider the patterns 'c' and 'cd' in Table 1. Given the user-defined* $per = 2$, $minPS = 3$ *and* $minRec = 2$, *the interesting periodic-intervals of 'c' and 'cd' are* $\{[2,12]\}$ *and* $\{[2,5],[9,12]\}$, *respectively. Therefore, the* $Rec(c) = |\{[2,12]\}| = 1$ *and* $Rec(cd) = |\{[2,5],[9,12]\}| = 2$. *As the* $Rec(c) \ngeq minRec$, *'c' is not a recurring pattern. However, its superset 'cd' is a recurring pattern because* $Rec(cd) \geq minRec$. *Thus, the recurring patterns do not satisfy the anti-monotonic property. The same can be observed in Table 2.*

We introduce the following pruning technique to reduce the computational cost of finding recurring patterns.

"Let $E_{rec}(X) = \sum_{i=1}^{|PS^X|} \left\lfloor \frac{ps_i^X}{minPS} \right\rfloor$. If $E_{rec}(X) < minRec$, then neither X nor its supersets will be recurring patterns"

The $E_{rec}(X)$ denotes the upper bound of *recurrence* that a superset of X can have in the database. Thus, we call $E_{rec}(X)$ the **estimated maximum recurrence of a superset of** X. The correctness of our pruning technique is straight forward to prove from Properties 1 and 2, and illustrated in Example 11.

PROPERTY 1. *For the pattern* X, $E_{rec}(X) \geq Rec(X)$.

PROPERTY 2. *If* $X \subset Y$, *then* $TS^X \supseteq TS^Y$ *and* $E_{rec}(X) \geq E_{rec}(Y)$.

EXAMPLE 11. *In Table 1, the item 'g' occurs in timestamps of* $1, 5, 6, 7, 12$ *and* 14. *Therefore,* $TS^g = \{1,5,6,7,$-$12,14\}$ *and* $S(g) = 6$. *If* $per = 2$, $minPS = 3$ *and* $minRec = 2$, *then* $TS_1^g = \{1\}$, $TS_2^g = \{5,6,7\}$, $TS_3^g = \{12,14\}$, $ps_1^g = |TS_1^g| = 1$, $ps_2^g = |TS_2^g| = 3$ *and* $ps_3^g = |TS_3^g| = 2$. *For this item,* $E_{rec}(g) = 1 \left(= \sum_{i=1}^{3} \left\lfloor \frac{ps_i^g}{3} \right\rfloor = \left\lfloor \frac{1}{3} \right\rfloor + \left\lfloor \frac{3}{3} \right\rfloor + \left\lfloor \frac{2}{3} \right\rfloor \right)$. *That is, any superset of 'g' can at most have recurrence value equal to 1, which is less than the user-defined minRec. Henceforth, pruning 'g' will not result in missing of any recurring pattern.*

Based on our proposed pruning technique, we introduce the following definition.

DEFINITION 11. ***(Candidate pattern*** X***.)*** *Pattern* X *is a candidate pattern if* $E_{rec}(X) \geq minRec$.

A candidate pattern containing only one item is called a **candidate item**. The candidate patterns satisfy the *anti-monotonic property* (Property 2). Therefore, we use candidate k-patterns to discover recurring $(k+1)$-patterns.

4.2 RP-growth: Structure, Construction and Mining

Traditional Frequent Pattern-growth algorithm [24] cannot be used for finding recurring patterns. This is because the structure of FP-tree captures only the frequency and disregards the periodic behavior of the patterns in a database. To address this issue, RP-growth introduces an alternative tree structure known as an Recurring Pattern-tree (RP-tree).

Our RP-growth algorithm involves the following two steps: (i) construction of an RP-tree and (ii) recursive mining of the RP-tree to discover the complete set of recurring patterns. Before we describe the above two steps, we introduce the structure of an RP-tree.

4.2.1 *Structure of RP-tree*

The structure of an RP-tree includes a prefix-tree and a candidate item list, called the RP-list. The RP-list consists of each distinct *item* (i) with *support* (s), *estimated maximum recurrence* (E_{rec}), and a pointer pointing to the first node in the prefix-tree carrying the item.

The prefix-tree in an RP-tree resembles the prefix-tree in FP-tree. However, to obtain both frequency and *periodic* behavior of the patterns, the nodes in an RP-tree explicitly maintain the occurrence information for each transaction by keeping an occurrence timestamp list, called a **ts-list**. To achieve memory efficiency, only the last node of every transaction maintains the ts-list. Hence, two types of nodes are maintained in a RP-tree: **ordinary node** and **tail-node**. The former is a type of node similar to that used in an FP-tree, whereas the latter represents the last item of any sorted transaction. Therefore, the structure of a *tail*-node is $i[ts_p, ts_q, ..., ts_r]$, $1 \leq p \leq q \leq r \leq m$, where i is the node's item name and ts_i, $i \in [1, m]$, is the timestamp of a transaction containing the items from *root* up to the node i. The conceptual structure of an RP-tree is shown in Figure 3. Like an FP-tree, each node in an RP-tree maintains parent, children, and node traversal pointers. Please note that no node in an RP-tree maintains the support count as in an FP-tree. To facilitate a high degree of compactness, items in the prefix-tree are arranged in support-descending order.

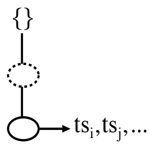


Figure 3: Conceptual structure of prefix-tree in RP-tree. Dotted ellipse represents ordinary node, while other ellipse represents tail-node of sorted transactions with timestamps $ts_i, ts_j \in R$

One can assume that the structure of the prefix-tree in an RP-tree may not be memory efficient since it explicitly maintains timestamps of each transaction. However, it has been argued that such a tree can achieve memory efficiency by keeping transaction information only at the *tail*-nodes and avoiding the support count field at each node [9]. Furthermore, an RP-tree avoids the *complicated combinatorial explosion problem of candidate generation* as in Apriori-like algorithms [23]. Keeping the information pertaining to transactional-identifiers in a tree can also be found in efficient frequent pattern mining [25].

Algorithm 1 RP-List(TDB: Transactional database, I: Set of items, *per*: period, $minPS$: minimum periodic-support, $minRec$: minimum recurrence)

```
1: Let $id_l$ be a temporary array that records the *timestamp* of the last appearance of each item in the $TDB$. Let $ps$ be another temporary array that records the periodic-support of an item in a subset of a database. Let $t = \{ts_{cur}, X\}$ denote the current transaction with $ts_{cur}$ and $X$ representing the timestamp of the current transaction and pattern, respectively.
2: for each transaction $t \in TDB$ do
3:   if an item $i$ occurs for the first time then
4:     Add $i$ to the RP-list.
5:     Set $s^i = 1$, $e^i_{rec} = 0$, $id^i_l = ts_{cur}$ and $ps^i = 1$.
6:   else
7:     if $(ts_{cur} - id^i_l) \leq per$ then
8:       Set $s^i++$, $ps^i++$ and $id^i_l = ts_{cur}$.
9:     else
10:      $e^i_{rec} += \left\lfloor \frac{ps^i}{minPS} \right\rfloor$.
11:      Set $s^i++$, $ps^i = 1$ and $id^i_l = ts_{cur}$. {Beginning of a new subset of a database.}
12:    end if
13:  end if
14: end for
15: To reflect the correct estimated recurrence value for each item in the RP-list, perform $e^i_{rec} += \left\lfloor \frac{ps^i}{minPS} \right\rfloor$.
```

Algorithm 2 RP-Tree(TDB, RF-list)

```
1: Create the root of an RP-tree, $T$, and label it "*null*".
2: for each transaction $t \in TDB$ do
3:   Set the timestamp of the corresponding transaction as $t_{cur}$.
4:   Select and sort the candidate items in $t$ according to the order of $CI$. Let the sorted candidate item list in $t$ be $[p|P]$, where $p$ is the first item and $P$ is the remaining list.
5:   Call $insert\_tree([p|P], t_{cur}, T)$.
6: end for
7: call RP-growth ($Tree$, $null$);
```

4.2.2 *Construction of RP-tree*

Since recurring patterns do not satisfy the *anti-monotonic property*, candidate 1-patterns (or items) will play an important role in effective mining of these patterns. The set of candidate items CI in a database for the user-defined *per*, $minPS$, and $minRec$ can be discovered by populating the RP-list with a scan on the database. Figure 4 shows the construction of an RP-list using Algorithm 1. Due to page limitation, we only present the key steps in the construction of the RP-list. Please note that the *per*, $minPS$, and $minRec$ values have been set to 2, 3 and 2, respectively.

The scan on the first transaction, "$1 : a, b, g$", with $ts_{cur} = 1$ initializes items 'a', 'b', and 'g' in the RP-list and sets their s, e_{rec}, id_l, and ps values to $1, 0, 1$, and 1, respectively (lines 1 to 5 in Algorithm 1). Figure 4(a) shows the RP-list generated after scanning the first transaction. The scan on the second transaction "$2 : a, c, d$" with $ts_{cur} = 2$ initializes the items 'c' and 'd' in the RP-list by setting their s, e_{rec}, id_l,

Algorithm 3 insert_tree($[p|P]$, t_{cur}, T)

```
1: while $P$ is non-empty do
2:   if $T$ has a child $N$ such that $p.itemName \neq N.itemName$ then
3:     Create a new node $N$. Let its parent link be linked to $T$. Let its node-link be linked to nodes with the same itemName via the node-link structure. Remove $p$ from $P$.
4:   end if
5: end while
6: Add $t_{cur}$ to the leaf node.
```

Algorithm 4 RP-growth($Tree$, α)

```
1: for each $a_i$ in the header of Tree do
2:   Generate pattern $\beta = a_i \cup \alpha$. Collect all of the $a_i's$ ts-lists into a temporary array, $TS^{\beta}$, and calculate $E^{\beta}_{rec}$.
3:   if $E^{\beta}_{rec} \geq minRec$ then
4:     Construct $\beta$'s conditional pattern base then $\beta$'s conditional RP-tree $Tree_{\beta}$. Call getRecurrence($\beta$, $TS^{\beta}$).
5:     if $Tree_{\beta} \neq \emptyset$ then
6:       call RP-growth($Tree_{\beta}$, $\beta$);
7:     end if
8:   end if
9:   Remove $a_i$ from the $Tree$ and push the $a_i$'s ts-list to its parent nodes.
10: end for
```

and *ps* values to 1, 0, 2, and 1, respectively. In addition, the s, e_{rec}, id_l, and *ps* values of an already existing item 'a' are updated to 2, 0, 2, and 2, respectively (lines 7 to 9 in Algorithm 1). Figure 4(b) shows the RP-list generated after scanning the second transaction. Figure 4(c) shows the RP-list constructed after scanning the seventh transaction. It can be observed that the 'e_{rec}' of 'a' and 'b' have been updated from 0 to 1. This is because their $\left\lfloor \frac{ps}{minPS} \right\rfloor = 1$ (line 10 in Algorithm 1). The *ps* value of 'a' and 'b' is set to 1 because they appeared periodically once again in the database (line 11 in Algorithm 1). Figure 4(d) shows the RP-list constructed after scanning every transaction in the database. The estimated recurrence (e_{rec}) value for all the items in the RP-list is once again computed to reflect the correctness (line 15 in Algorithm 1). Figure 4(e) shows the updated e_{rec} value for all items in the RP-list. Using our pruning technique, 'g' is removed from the RP-list as its $e_{cur} < minRec$. The remaining items are sorted in descending order of their support values (line 16 in Algorithm 1). Figure 4(f) shows the sorted list of candidate items in the RP-list.

After finding candidate items, we conduct another scan on the database and construct the prefix-tree of the RP-tree, as in Algorithms 2 and 3. These procedures are the same as those for constructing an FP-tree [24]. However, the major difference is that no node in an RF-tree maintains the *support* count, as in an FP-tree. The first transaction $\{1 : a, b, g\}$ is scanned and a branch is constructed in the RP-tree with only the candidate items 'b' and 'a.' The *tail* node 'b : 1' carries the *timestamp* of the transaction. The RP-tree generated after scanning the first transaction is shown in Figure 5(a). A similar process is repeated for the remaining transactions and the tree is updated accordingly. Figure 5(b) shows the RP-tree constructed after scanning the entire database. For simplicity, we do not show the node traversal pointers in trees; however, they are maintained like an FP-tree does.

(a)

i	s	e_{rec}	ps	id_l
a	1	0	1	1
b	1	0	1	1
g	1	0	1	1

(b)

i	s	e_{rec}	ps	id_l
a	2	0	2	2
b	1	0	1	1
g	1	0	1	1
c	1	0	1	2
d	1	0	1	2

(c)

i	s	e_{rec}	ps	id_l
a	5	1	1	7
b	4	1	1	7
g	4	0	3	7
c	4	0	4	7
d	3	0	3	5
e	3	0	3	6
f	3	0	3	6

(d)

i	s	e_{rec}	ps	id_l
a	8	1	3	14
b	7	1	3	14
g	6	1	2	14
c	7	0	7	12
d	6	1	3	12
e	6	1	3	12
f	6	1	3	12

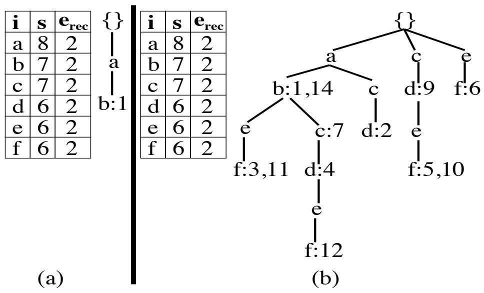
(e)

i	s	e_{rec}
a	8	2
b	7	2
g	6	1
c	7	2
d	6	2
e	6	2
f	6	2

(f)

i	s	e_{rec}
a	8	2
b	7	2
c	7	2
d	6	2
e	6	2
f	6	2

Figure 4: Construction of RP-list: (a) after scanning first transaction, (b) after scanning second transaction, (c) after scanning seventh transaction, (d) after scanning every transaction, (e) after calculating actual 'e_{rec}' values, and (f) sorted list of candidate items



i	s	e_{rec}
a	8	2
b	7	2
c	7	2
d	6	2
e	6	2
f	6	2

i	s	e_{rec}
a	8	2
b	7	2
c	7	2
d	6	2
e	6	2
f	6	2

Figure 5: Construction of RP-tree: (a) after scanning first transaction and (b) after scanning entire transactional database

The RP-tree maintains the complete information of all recurring patterns in a database. The correctness is based on Property 3 and shown in Lemmas 1 and 2. For each transaction $t \in theTDB$, $CI(t)$ is the set of all candidate items in t, i.e., $CI(t) = item(t) \cap CI$, and is called the candidate item projection of t.

PROPERTY 3. *An RP-tree maintains a complete set of candidate item projections for each transaction in a database only once.*

LEMMA 1. *Given a TDB and user-defined per, (minPS), and minRec values, the complete set of all recurring item projections of all transactions in the TDB can be derived from the RP-tree.*

PROOF. Based on Property 3, each transaction $t \in TDB$ is mapped to only one path in the tree, and any path from the *root* up to a *tail* node maintains the complete projection for exactly n transactions (where n is the total number of entries in the *ts*-list of the *tail* node). □

Lemma 2. *The size of the RP-tree (without the root node) on a TDB for user-defined per, minPS, and minRec is bounded by* $\sum_{t \in TDB} |CI(t)|$.

Proof. According to the RP-tree construction process and Lemma 1, each transaction t contributes at most one path of size $|CI(t)|$ to an RP-tree. Therefore, the total size contribution of all transactions can be $\sum_{t \in TDB} |CI(t)|$ at best. However, since there are usually many common prefix patterns among the transactions, the size of an RP-tree is normally much smaller than $\sum_{t \in TDB} |CI(t)|$. □

Algorithm 5 getRecurrence(X: pattern, TS^X:ts-list of pattern X)

```
1: Let $id_l$ be a variable that records the timestamp of the last transaction containing $X$. Let $subDB$ be a list of pairs of the form $(startTS, endTS)$, where $startTS$ and $endTS$ respectively represent the starting and ending timestamps of periodic appearances of a pattern in a subset of data. It is used to record the periodic-intervals of a pattern. Let $currentPS$ be a variable to measure the periodic-support of $X$ in a periodic-interval.
2: for each timestamp $ts_{cur} \in TS^X$ do
3:   if ($ts_{cur}$ is $X$'s first occurrence) then
4:     $currentPS = 1$, $startTS = ts_{cur}$;
5:   else
6:     if ($ts_{cur} - id_l \leq per$) then
7:       $currentPS++$;
8:     else
9:       if ($currentPS \geq minPS$) then
10:        $subDB.insert(startTS, id_l)$;
11:      end if
12:      $currentPS = 1$, $startTS = ts_{cur}$;
13:    end if
14:  end if
15:  $id_l = ts_{cur}$;
16: end for
17: // To reflect correct recurrence of $X$.
18: if ($currentPS \geq minPS$) then
19:   $subDB.insert(startTS, id_l)$;
20: end if
21: return (($subDB.size() \geq minRec$)?true:false);
```

4.2.3 Mining Recurring Patterns

Although an RP-tree and FP-tree arrange items in support-descending order, we cannot directly apply FP-growth mining on an RP-tree. The reasons are as follows: (*i*) an RP-tree does not maintain the support count at each node, and it handles the *ts*-lists at the *tail* nodes and (*ii*) recurring patterns do not satisfy the *anti-monotonic property.* We devised another pattern growth-based bottom-up mining technique to mine the patterns. The basic operations in mining an RP-tree includes: (*i*) counting length-1 candidate items, (*ii*) constructing the prefix tree from each candidate pattern, and (*iii*) constructing the conditional tree from each prefix-tree. The RP-list provides the length-1 candidate items. Before we discuss the prefix-tree construction process, we explore the following important property and lemma of an RP-tree.

Property 4. *A tail node in an RP-tree maintains the occurrence information for all the nodes in the path (from the tail node to the root) at least in the transactions in its ts-list.*

Lemma 3. *Let* $Z = \{a_1, a_2, \cdots, a_n\}$ *be a path in an RP-tree where node* a_n *is the tail node carrying the ts-list of the path. If the ts-list is pushed-up to node* a_{n-1}*, then* a_{n-1} *maintains the occurrence information of the path* $Z' = \{a_1, a_2, \cdots, a_{n-1}\}$ *for the same set of transactions in the ts-list without any loss.*

Proof. Based on Property 4, a_n maintains the occurrence information of path Z' at least in the transactions in its *ts*-list. Therefore, the same *ts*-list at node a_{n-1} maintains the same transaction information for Z' without any loss. □

The procedure to discover recurring patterns from RP-tree is shown in Algorithm 4. The working of this algorithm is as follows. We proceed to construct the prefix tree for each candidate item in the RP-list, starting from the bottom-most item, say i. To construct the prefix-tree for i, the prefix sub-paths of nodes i are accumulated in a tree-structure, PT_i. Since i is the bottom-most item in the RP-list, each node labeled i in the RP-tree must be a tail node. While constructing PT_i, based on Property 4, we map the *ts*-list of every node of i to all items in the respective path explicitly in the temporary array (one for each item). This temporary array facilitates the calculation of *support* and e_{rec} of each item in PT_i (line 2 in Algorithm 4). If an item j in PT_i has *support* $\geq$ *minSup* and $e_{rec} \geq$ *minRec*, then we construct its conditional tree and mine it recursively to discover the recurring patterns (lines 3 to 8 in Algorithm 4). Moreover, to enable the construction of the prefix-tree for the next item in the RP-list, based on Lemma 3, the *ts*-lists are pushed-up to the respective parent nodes in the original RP-tree and in PT_i as well. All nodes of i in the original RP-tree and i's entry in the RP-list are deleted thereafter (line 9 in Algorithm 4).

Using Properties 1 and 2, the conditional tree CT_i for PT_i is constructed by removing all those items from PT_i that have $e_{rec} <$ *minRec.* If the deleted node is a *tail* node, its *ts*-list is pushed-up to its parent node. The contents of the temporary array for the bottom item j in the RP-list of CT_i represent TS^{ij} (i.e., the set of all timestamps where items i and j have appeared together in the database). Therefore, using Algorithm 5, the *recurrence* of "ij" is computed and it is determined whether "ij" is a recurring pattern. The same process of creating a prefix-tree and its corresponding conditional tree is repeated for further extensions of "ij". The whole process of mining for each item is repeated until *RP-list* $\neq \emptyset$.

Consider item 'f', which is the last item in the RP-list in Figure 4(e). The prefix-tree for 'f', PT_f, is constructed from the RP-tree, as shown in Figure 6(a). There are five items, 'a, b, c, d', and 'e' in PT_f. Only item 'e' satisfies the condition $E_{rec}(e) \geq minRec$. Therefore, the conditional tree CT_f from PT_f is constructed with only one item 'e', as shown in 6(b). The *ts*-list of 'e' in CT_f' generates TS^{ef}. The "*recurrence*" of 'ef' is measured using Algorithm 5. Since $Rec(ef) \geq minRec$, 'ef' will be generated as a recurring pattern. A similar process is repeated for the other items in the RP-list. Next, 'f' is pruned from the original RP-tree

Table 4: **User-defined *per*, *minPS* and *minRec* values in different databases**

	per			*minPS*			*minRec*		
	1	2	3	1	2	3	1	2	3
T10I4D100k	360	720	1440	0.1%	0.2%	0.3%	1	2	3
Shop-14	360 (=6 hr.)	720 (=12 hrs.)	1440 (=24 hrs.)	0.1%	0.2%	0.3%	1	2	3
Twitter	360 (=6 hr.)	720 (=12 hrs.)	1440 (=24 hrs.)	2%	5%	10%	1	2	3

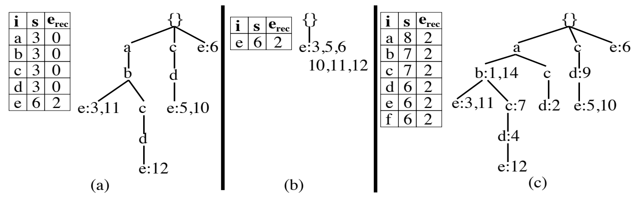


Figure 6: Finding RP-patterns for suffix item '*f*' in RP-tree: (a) prefix-tree for item '*f*' (i.e., PT_f), (b) conditional tree for item '*f*' (i.e., CT_f), and (c) RP-tree after pruning item '*f*'

and its *ts*-lists are pushed to its parent nodes, as shown in 6(c). All the above processes are once again repeated until the RP-list $\neq \emptyset$.

The above bottom-up mining technique on a support descending RP-tree is efficient, because it shrinks the search space dramatically as the mining process progresses.

5. EXPERIMENTAL RESULTS

In this section, we first evaluate the performance of RP-growth. Next, we discuss the usefulness of recurring patterns by comparing them against p-patterns and periodic-frequent patterns. It should be noted that our recurring patterns are the generalization of periodic-frequent patterns, as the latter patterns exhibit complete (rather than partial) cyclic repetitions in the entire database. There are only two Apriori-like algorithms, periodic-first and association-first, to discover p-patterns. We use the periodic-first algorithm to discover p-patterns since it is relatively faster than the association-first algorithm. We use the Periodic-Frequent pattern-growth++ algorithm (PF-growth++) [15] to discover periodic-frequent patterns. We do not compare the performance of RP-growth against the periodic-first and PF-growth++ algorithms. The reason is that the other algorithms discover regular patterns; therefore, they use different measures to assess the periodic interestingness of a pattern.

5.1 Experimental setup

The algorithms, RP-growth, periodic-first and PF-growth++, were written in GNU C++ and run with Ubuntu 14.4 on a 2.66 GHz machine with 8 GB of memory. To the best of our knowledge, there are no publicly available time-based sequences. Therefore, we conducted experiments using the following databases.

- **T10I4D100K database.** This database is a synthetic transactional database generated using the procedure given by [23]. This database contains 100,000 transactions and 941 distinct items.
- **Shop-14 database.** A Czech company provided click-stream data of seven online stores in the ECML/PKDD 2005 Discovery challenge. We considered the click stream data of product categories visited by the users in "Shop 14" (www.shop4.cz), and created a transactional database with each transaction representing the set of web pages visited by the people at a particular *minute interval.* The transactional database contains 59,240 transactions (i.e., 41 days of page visits) and 138 distinct items (or product categories).
- **Twitter database.** We created this database by considering the top 1000 English hashtags appearing in 44 million tweets/retweets from 1-May-2013 to 31-August-2013 (i.e., 123 days). The measure, *term frequency-inverse document frequency*, is used to rank the hashtags. The timestamp of each transaction represents a minute starting from 00:00 hours of 1-May-2013 to 24:00 hours of 31-August-2013. The resultant transactional database has 177,120 transactions with 1000 distinct items (or hashtags). More details on the data collection process and the usefulness of this data in finding interesting events has been presented in [26].

To mine p-patterns and recurring patterns, we transformed all the above databases into time-based sequences using the timestamps of each transaction. As this transformation process is a rather simple and straight-forward approach, we do not discuss it for brevity.

Table 4 lists the different *per*, *minPS* and *minRec* values used in above the databases. The periodic interval (i.e., *per* value) in both Shop-14 and Twitter databases varied from six hours to one day. Similarly, the *minRec* in these databases varied from 1 to 3. In this paper, we do not present the results for *minRec* values greater than 3 because very few recurring patterns were getting generated at *minRec* > 3. The *minPS* values in T10I4D100K and Shop-14 databases varied from 0.1% to 0.3%. The reason for choosing low *minPS* values is to discover the patterns involving both frequent and rare items. In the Twitter database, we set *minPS* at 2%, 5% and 10%. The reason for choosing a relatively high *minPS* values as compared with the other two databases is that very low *minPS* values are resulting in a combinatorial explosion producing too many patterns.

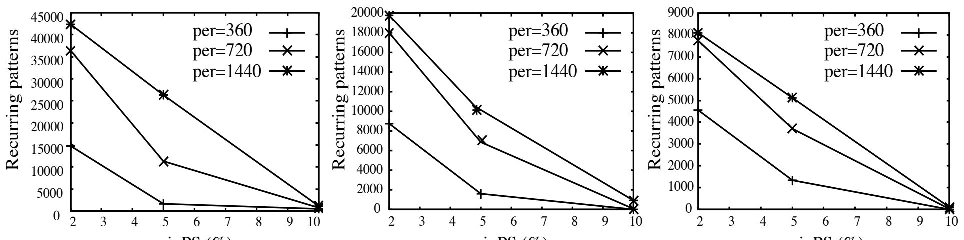
5.2 Generation of recurring patterns

Table 5 lists the numbers of recurring patterns discovered in T10I4D100K, Shop-14 and Twitter databases at different *per*, *minPS* and *minRec* values. The partial results of Table 5 are shown in Figure 7. The following observations can drawn from this figure:

- At a fixed *per* and *minRec*, the increase in *minPS* can decrease the number of recurring patterns. The reason is that many patterns failed to appear periodically for longer time periods.

Table 5: Number of recurring patterns generated at different *per*, *minPS* and *minRec* threshold values

Dataset	*minPS*	Number of recurring patterns								
		minRec = 1			*minRec* = 2			*minRec* = 3		
		per=360	*per*=720	*per*=1440	*per*=360	*per*=720	*per*=1440	*per*=360	*per*=720	*per*=1440
T10I4-D100k	0.1%	428	1254	7193	255	436	1036	194	160	27
	0.2%	339	757	3205	168	103	39	72	0	0
	0.3%	296	622	2148	109	32	2	21	0	0
Shop-14	0.1%	593	1885	4977	447	1339	3198	338	266	9
	0.2%	342	1077	1906	257	750	1470	118	14	0
	0.3%	251	744	933	195	534	760	48	3	0
Twitter	2%	14736	36354	42319	8718	17982	19746	4551	7749	8103
	5%	1655	11268	26341	595	6847	7010	337	3713	5123
	10%	511	714	1190	11	34	912	6	17	98



(a) minRec=1 (b) minRec=2 (c) minRec=3

Figure 7: Recurring patterns discovered in Twitter data

- At a fixed *minPS* and *per*, the increase in *minRec* can decrease the number of recurring patterns. This is because many patterns failed to satisfy the increased *minRec* values.
- At a fixed *minPS*, the increase in *per* can have different impact on the generation of recurring patterns for the values $minRec = 1$ and $minRec > 1$. At $minRec = 1$, increase in *per* can increase the number of recurring patterns. The reason is that the inter-arrival times of the patterns that were considered as aperiodic at low *per* values were considered as periodic with the increase in the *per* value. For $minRec > 1$, increase in *per* can either increase or decrease the number of recurring patterns. The reason for decrease is due to the merging of interesting periodic-intervals discovered at low *per* values.

Table 6 lists some of the recurring patterns discovered from the Twitter database at $per = 360$, $minPS = 2\%$ and $minRec = 1$. Figures 8 (a) and (b) show the frequencies of the terms present in patterns {yyc, uttarakhand} and {nuclear, hibaku} on a daily basis. It can be observed that "uttarakhand" is a relatively rare term as compared with other terms, and our model has effectively discovered the knowledge pertaining to this term. Another interesting observation from Table 5 is that even at low *minPS* values, our model has generated only a limited number of recurring patterns in each database. This clearly shows that our model can discover the knowledge pertaining to rare terms without producing too many uninteresting patterns. In other words, our model is tolerant of the "rare item problem".

5.3 Performance of RP-growth

Table 7 lists the runtime required using RP-growth to discover recurring patterns in T10I4D100K, Shop-14, and

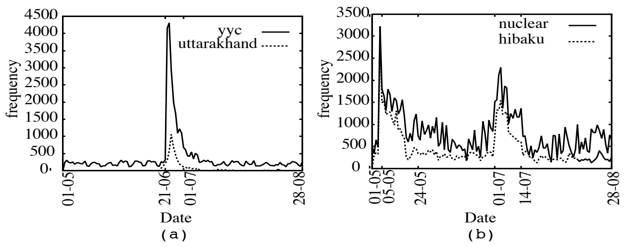


Figure 8: Frequency of hashtags at different days in database. Date is of form 'dd-mm'. Year of this date is 2013

Twitter databases. The runtime involves the time taken to transform the time-based sequence into a transactional database and mining of recurring patterns. Figure 9 shows the runtime required by RP-growth while mining the recurring patterns in Twitter database. The changes in the *per*, *minPS* and *minRec* threshold values shows a similar effect on runtime consumption as in the generation of recurring patterns. The proposed algorithm discovered the complete set of recurring patterns at a reasonable runtime even at low *minPS* thresholds.

5.4 Comparison of p-patterns, recurring patterns and periodic-frequent patterns

We compared p-patterns, recurring patterns and periodic-frequent patterns at different *per* and *minPS* values. For brevity, we present the results discovered when *period* is set to 1 day, i.e., $per = 1440$. The *minSup* and *minPS* values are set to 0.1% and 2% respectively for Shop-14 and Twitter databases. The p-pattern mining requires another parameter known as window length (w). We set $w = 1$ for our experiment.

Table 6: Some of the interesting recurring patterns discovered in Twitter database

S.No	Pattern	Periodic duration	Cause for the events
1	{yyc, uttarakhand}	[2013-06-21 01:08, 2013-07-01 04:27]	On June 20, Uttarakhand, a state in India and Alberta, a province in Canada have witnessed heavy floods.
2	{nuclear, hibaku} (In Japanese, hibaku means radiation.)	[2013-05-06 22:33, 2013-05-24 22:13], [2013-07-01 06:17, 2013-07-14 06:21]	(*i*) A Japanese minister has visited Chernobyl, Ukraine to learn from the recovery from the severe nuclear accident. (*ii*) People were tweeting about detection of Plutonium at a point 12 KM from Fukoshima nuclear reactor.
3	{pakvotes, nayapakistan}	[2013-05-09 16:15, 2013-05-15 14:11]	The general elections were held in Pakisthan. on May 11, 2013.
4	{oklahoma, tornado, prayforoklahoma}	[2013-05-21 11:52, 2013-05-24 21:38]	Oklahoma was struck with a tornado on May 20, 2013.

Table 7: Runtime of RP-growth at different *per*, *minPS* and *minRec* threshold values

Dataset	*minPS*	Runtime of RP-growth								
		minRec = 1			*minRec* = 2			*minRec* = 3		
		per=360	*per*=720	*per*=1440	*per*=360	*per*=720	*per*=1440	*per*=360	*per*=720	*per*=1440
T10I4-D100k	0.1%	14.8	150.9	366.5	3.8	10.7	40.1	3.5	3.9	6.3
	0.2%	7.7	45.9	99.6	3.6	5.4	9.6	2.7	3.1	3.1
	0.3%	3.7	11.6	21.3	3.2	3.4	4.2	2.5	2.4	2.6
Shop-14	0.1%	47.7	55.6	67.3	43.5	47.7	52.3	42.4	45.1	48.2
	0.2%	42.9	46.1	51.3	41.7	43.4	45.0	41.4	42.1	43.8
	0.3%	42.4	44.0	47.3	41.6	42.1	43.6	41.1	41.5	41.7
Twitter	2%	55.1	190.0	290.5	42.9	154.9	248.4	41.3	139.2	226.1
	5%	37.9	134.3	225.6	33.0	105.3	181.9	31.5	96.1	159.7
	10%	32.3	108.3	190.9	30.4	89.2	151.3	29.9	66.9	124.1

Table 8 lists the numbers of periodic-frequent patterns, recurring patterns and p-patterns discovered in the Shop-14 and Twitter databases. The column labeled '*II*' in this table refers to the length of the longest pattern discovered in each of these databases. The following observations can be drawn from this table.

First, the total numbers of periodic-frequent patterns discovered in both databases were relatively less than the number of recurring patterns and p-patterns. The reason is that the strict constraint that a frequent pattern has to exhibit complete cyclic repetitions throughout the data has failed to identify many interesting partial periodically appearing patterns. Moreover, this strict constraint also resulted in finding the very short periodic-frequent patterns (see columns labeled '*II*' in Table 8). This is because longer patterns generally fail to exhibit complete cyclic repetitions throughout the data. Setting a high *period* threshold value can enable a user to discover long periodic-frequent patterns. However, this high *period* can result in discovering sporadically appearing patterns as periodic-frequent patterns. Thus, it is necessary to relax this strict constraint without changing the *period* threshold value. As our model enables a user to relax this strict constraint, recurring patterns have been found more interesting than the periodic-frequent patterns for a given *per* threshold.

Second, the total number of p-patterns discovered in both databases were much higher than the recurring patterns and periodic-frequent patterns. The reason is that the usage of a low *minSup* has facilitated Ma and Hellerstein's model [7] to discover not only all our recurring patterns as p-patterns, but also resulted in a combinatorial explosion of frequently appearing items producing too many p-patterns. Most importantly, many of the p-patterns discovered at low *minSup* were uninteresting to the users. The reason is that frequent

Table 8: Number of patterns discovered in Shop-14 and Twitter databases. Terms '*I*' and '*II*' represent *total number of patterns* and *maximum length of pattern* found in each database, respectively.

	Shop-14		Twitter	
	I	*II*	*I*	*II*
PF patterns	22	3	466	2
Recurring patterns	4,977	9	42,319	7
p-patterns	156,7001	12	442,076	16

items were combining with one another in all possible ways and generating p-patterns by satisfying a low *minSup* value. On the contrary, our model has reduced the combinatorial explosion of frequent items by assessing their interestingness with respect to their number of consecutive periodic appearances in a portion of data.

6. CONCLUSIONS AND FUTURE WORK

We introduced a new class of partial periodic patterns known as *recurring patterns* and discussed the usefulness of these patterns in various real-world applications. We also proposed a model for discovering such patterns. The patterns discovered with the proposed model do not satisfy the anti-monotonic property. Therefore, we proposed a novel pruning technique to reduce the computational cost of finding these patterns. We also proposed a pattern-growth algorithm to discover the recurring patterns effectively. Experimental results suggest that the model is tolerant to the "rare item problem" and that the algorithm is efficient. The usefulness of the recurring patterns was discussed by comparing them against the periodic-frequent and p-patterns.

In our current study, we did not considered noisy data and the phase-shifts of the items within the data. For fu-

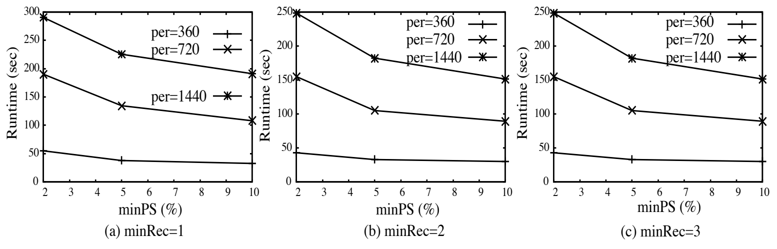


Figure 9: Runtime of RP-growth

ture work, we will develop methods for handling these two scenarios. Another interesting future work will be extending our model to improve the performance of an association rule-based recommender system.

7. ACKNOWLEDGMENTS

This work was supported by the Research and Development on Real World Big Data Integration and Analysis program of the Ministry of Education, Culture, Sports, Science, and Technology, JAPAN.

8. REFERENCES

[1] C. M. Antunes and A. L. Oliveira, "Temporal data mining: An overview," in *Workshop on Temporal Data Mining, KDD*, 2001.

[2] B. Özden, S. Ramaswamy, and A. Silberschatz, "Cyclic association rules," in *ICDE*, 1998, pp. 412–421.

[3] Z. Li, B. Ding, J. Han, R. Kays, and P. Nye, "Mining periodic behaviors for moving objects," in *KDD '10*, 2010, pp. 1099–1108.

[4] P. Esling and C. Agon, "Time-series data mining," *ACM Computing Surveys*, vol. 45, no. 1, pp. 12:1–12:34, Dec. 2012.

[5] J. Han, W. Gong, and Y. Yin, "Mining segment-wise periodic patterns in time-related databases." in *KDD*, 1998, pp. 214–218.

[6] J. Han, G. Dong, and Y. Yin, "Efficient mining of partial periodic patterns in time series database," in *ICDE*, 1999, pp. 106–115.

[7] S. Ma and J. Hellerstein, "Mining partially periodic event patterns with unknown periods," in *ICDE*, 2001, pp. 205–214.

[8] R. Yang, W. Wang, and P. Yu, "Infominer+: mining partial periodic patterns with gap penalties," in *ICDM*, 2002, pp. 725–728.

[9] S. K. Tanbeer, C. F. Ahmed, B. S. Jeong, and Y. K. Lee, "Discovering periodic-frequent patterns in transactional databases," in *PAKDD*, 2009, pp. 242–253.

[10] C. Berberidis, I. Vlahavas, W. Aref, M. Atallah, and A. Elmagarmid, "On the discovery of weak periodicities in large time series," in *PKDD*, 2002, pp. 51–61.

[11] H. Cao, D. Cheung, and N. Mamoulis, "Discovering partial periodic patterns in discrete data sequences," in *Advances in Knowledge Discovery and Data Mining*, 2004, vol. 3056, pp. 653–658.

[12] W. G. Aref, M. G. Elfeky, and A. K. Elmagarmid, "Incremental, online, and merge mining of partial periodic patterns in time-series databases," *IEEE TKDE*, vol. 16, no. 3, pp. 332–342, Mar. 2004.

[13] B. Liu, W. Hsu, and Y. Ma, "Mining association rules with multiple minimum supports," in *KDD*, 1999, pp. 337–341.

[14] R. U. Kiran and P. K. Reddy, "Towards efficient mining of periodic-frequent patterns in transactional databases," in *DEXA (2)*, 2010, pp. 194–208.

[15] R. U. Kiran and M. Kitsuregawa, "Novel techniques to reduce search space in periodic-frequent pattern mining," in *DASFAA (2)*, 2014, pp. 377–391.

[16] M. M. Rashid, M. R. Karim, B. S. Jeong, and H. J. Choi, "Efficient mining regularly frequent patterns in transactional databases," in *DASFAA (1)*, 2012, pp. 258–271.

[17] J. Yang, W. Wang, and P. Yu, "Mining asynchronous periodic patterns in time series data," *IEEE TKDE*, vol. 15, no. 3, pp. 613–628, May 2003.

[18] C. H. Mooney and J. F. Roddick, "Sequential pattern mining – approaches and algorithms," *ACM Comput. Surv.*, vol. 45, no. 2, pp. 19:1–19:39, Mar. 2013.

[19] H. Mannila, "Methods and problems in data mining," in *ICDT*, 1997, pp. 41–55.

[20] S. Laxman, P. S. Sastry, and K. P. Unnikrishnan, "A fast algorithm for finding frequent episodes in event streams," in *KDD*, 2007, pp. 410–419.

[21] T. Oates, "Peruse: An unsupervised algorithm for finding recurrig patterns in time series," in *ICDM*, 2002, pp. 330–337.

[22] Y. Mohammad and T. Nishida, "Approximately recurring motif discovery using shift density estimation," in *Recent Trends in Applied Artificial Intelligence*, 2013, pp. 141–150.

[23] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," in *SIGMOD*, 1993, pp. 207–216.

[24] J. Han, J. Pei, Y. Yin, and R. Mao, "Mining frequent patterns without candidate generation: A frequent-pattern tree approach," *Data Min. Knowl. Discov.*, vol. 8, no. 1, pp. 53–87, Jan. 2004.

[25] M. J. Zaki and C.-J. Hsiao, "Efficient algorithms for mining closed itemsets and their lattice structure," *IEEE Trans. on Knowl. and Data Eng.*, vol. 17, no. 4, pp. 462–478, Apr. 2005.

[26] G. Rattanaritnont, M. Toyoda, and M. Kitsuregawa, "Analyzing patterns of information cascades based on users' influence and posting behaviors," in *TempWeb '12*, 2012, pp. 1–8.

Learning Path Queries on Graph Databases

Angela Bonifati   Radu Ciucanu   Aurélien Lemay
University of Lille & INRIA, France
{angela.bonifati, radu.ciucanu, aurelien.lemay}@inria.fr

ABSTRACT

We investigate the problem of learning graph queries by exploiting user examples. The input consists of a graph database in which the user has labeled a few nodes as *positive* or *negative examples*, depending on whether or not she would like the nodes as part of the query result. Our goal is to handle such examples to find a query whose output is what the user expects. This kind of scenario is pivotal in several application settings where unfamiliar users need to be assisted to specify their queries. In this paper, we focus on *path queries* defined by *regular expressions*, we identify fundamental difficulties of our problem setting, we formalize what it means to be *learnable*, and we prove that the class of queries under study enjoys this property. We additionally investigate an interactive scenario where we start with an empty set of examples and we identify the *informative nodes* i.e., those that contribute to the learning process. Then, we ask the user to label these nodes and iterate the learning process until she is satisfied with the learned query. Finally, we present an experimental study on both real and synthetic datasets devoted to gauging the effectiveness of our learning algorithm and the improvement of the interactive approach.

1. INTRODUCTION

Graph databases [41] are becoming pervasive in several application scenarios such as the Semantic Web [5], social [37] and biological [36] networks, and geographical databases [2], to name a few. A graph database is essentially a directed, edge-labeled graph. As an example, consider in Figure 1 a graph representing a geographical database having as nodes the neighborhoods of a city area (N_1 to N_6), along with cinemas (C_1 and C_2), and restaurants (R_1 and R_2) in such neighborhoods. The edges represent public transportation facilities from a neighborhood to another (using labels *tram* and *bus*), along with other kind of facilities (using labels *cinema* and *restaurant*). For instance, the graph indicates that one can travel by bus between the neighborhoods N_2 and N_3, that in the neighborhood N_4 exists a cinema C_1, and so on.

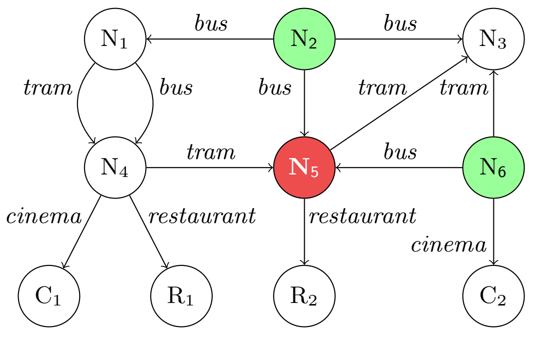


Figure 1: A geographical graph database.

Many mechanisms have been proposed to query a graph database, the majority of them being based on *regular expressions* [7, 41]. By continuing on our running example, imagine that a user wants to know from which neighborhoods in the city represented in Figure 1 she can reach cinemas via public transportation. These neighborhoods can be retrieved using a *path query* defined by the following *regular expression*:

$$q = (tram + bus)^* \cdot cinema$$

The query q selects the nodes N_1, N_2, N_4, and N_6 as they are entailed by the following *paths* in the graph:

$$N_1 \xrightarrow{tram} N_4 \xrightarrow{cinema} C_1,$$
$$N_2 \xrightarrow{bus} N_1 \xrightarrow{tram} N_4 \xrightarrow{cinema} C_1,$$
$$N_4 \xrightarrow{cinema} C_1,$$
$$N_6 \xrightarrow{cinema} C_2.$$

Although very expressive, graph query languages are difficult to understand by non-expert users who are unable to specify their queries with a formal syntax. The problem of *assisting non-expert users to specify their queries* has been recently raised by Jagadish et al. [24, 34]. More concretely, they have observed that "constructing a database query is often challenging for the user, commonly takes longer than the execution of the query itself, and does not use any insights from the database". While they have mentioned these problems in the context of relational databases, we argue that they become even more difficult to tackle for graph databases. Indeed, graph databases usually do not carry proper metadata as they lack schemas and/or do not ex-

10.5441/002/edbt.2015.11

hibit a clear distinction between instances and schemas. The absence of metadata along with the difficulty of visualizing possibly large graphs make unfeasible traditional query specification paradigms for non-expert users, such as query by example [43]. Our work follows the recent trend of specifying graph queries by example [33, 25]. Precisely, we focus on graph queries using regular expressions, which are fundamental building blocks of graph query languages [7, 41], while both [33, 25] consider simple graph patterns.

While the problem of executing path queries defined by regular expressions on graphs has been extensively studied recently [6, 32, 27], no research has been done on how to actually specify such queries. Our work focuses on the problem of assisting non-expert users to specify such path queries, by exploiting elementary user input.

By continuing on our running example, we assume that the user is not familiar with any formal syntax of query languages, while she still wants to specify the above query q on the graph database in Figure 1 by providing examples of the query result. In particular, she would positively or negatively label some graph nodes according to whether or not they would be selected by the targeted query. Thus, let us imagine that the user labels the nodes N_2 and N_6 as *positive examples* because she wants these nodes as part of the result. Indeed, one can reach cinemas from N_2 and N_6, respectively, through the following paths:

$$\mathrm{N}_2 \xrightarrow{bus} \mathrm{N}_1 \xrightarrow{tram} \mathrm{N}_4 \xrightarrow{cinema} \mathrm{C}_1,$$

$$\mathrm{N}_6 \xrightarrow{cinema} \mathrm{C}_2.$$

Similarly, the user labels the node N_5 as a *negative example* since she would not like it as part of the query result. Indeed, there is no path starting in N_5 through which the user can reach a cinema. We also observe that the query q above is *consistent* with the user's examples because q selects all positive examples and none of the negative ones. Unfortunately, there may exist an infinite number of queries consistent with the given examples. Therefore, we are interested to find either the "exact" query that the user has in mind or, alternatively, an equivalent query, which is close enough to the user's expectations.

Apart from assisting unfamiliar users to specify queries, our research has other crucial applications, such as mining *scientific workflows*. Regular expressions have already been used in the literature as a well-suited mechanism for inter-workflow coordination [21]. The path queries on graph databases that we study in this paper can be applied to assist scientists in identifying interrelated workflows that are of interest for them. For instance, assume that a biologist is interested in retrieving all interrelated workflows having a pattern that starts with protein purification, continues with an arbitrary number of protein separation steps, and ends with mass spectrometry. This corresponds to the following regular expression:

$$\mathit{ProteinPurification} \cdot \mathit{ProteinSeparation}^* \cdot \mathit{MassSpectrometry}.$$

Instead of specifying such a pattern in a formal language, the biologist may be willing to label some sequences of modules from a set of available workflows as positive or negative examples, as illustrated in Figure 2. Our algorithms can be thus applied to infer the workflow pattern that the biologist has in mind. Typically in graphs representing workflows the labels are attached to the nodes (e.g., as in Figure 2) instead of the edges. In this paper, we have opted for

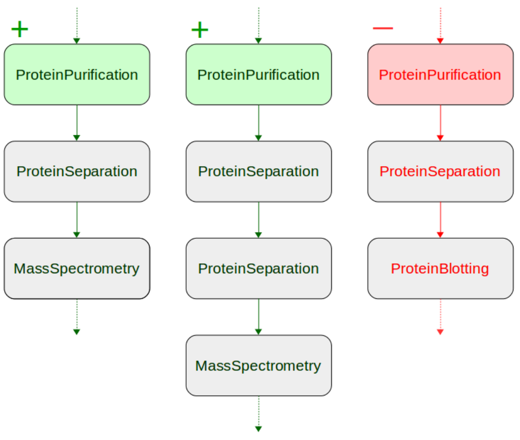


Figure 2: A set of scientific workflows examples.

edge-labeled graphs rather than node-labeled graphs, but our algorithms and learning techniques for the considered class of queries are applicable to the latter class of graphs in a seamless fashion. The problem of mining scientific workflows has been considered in recent work [8], which leverages data instances as representatives of the input and output of a workflow module. In our approach, we rely on simpler user feedback, namely Boolean labeling of sequences of modules across interrelated workflows.

Since our goal is to infer the user queries while minimizing the amount of user feedback, our research is also applicable to *crowdsourcing* scenarios [19], in which such minimization typically entails lower financial costs. Indeed, we can imagine that crowdworkers provide the set of positive and negative examples mentioned above for path query learning. Moreover, our work can be used in assisting non-expert users in other fairly complex tasks, such as specifying *schema mappings* [42] i.e., logical assertions between two path queries, one on a source schema and another on a target schema.

To the best of our knowledge, our work is the first to study the problem of learning path queries defined by regular expressions on graphs via user examples. More precisely, we make the following main contributions:

- We investigate a learning framework inspired by computational learning theory [26], in particular by grammatical inference [18] and we identify fundamental difficulties of such a framework. We consequently propose a definition of learnability adapted to our setting.
- We propose a learning algorithm and we precisely characterize the conditions that a graph must satisfy to guarantee that every user's goal query can be learned. Essentially, the main theoretical result of the paper states that for every query q there exists a polynomial set of examples that given as input to our learning algorithm guarantees the learnability of q. Additionally, our learning algorithm is guaranteed to run in polynomial time, whether or not the aforementioned set of examples is given as input.
- We investigate an interactive scenario, which bootstraps with an empty set of examples and builds it along the way. Indeed, the learning algorithm finely

interacts with the user by proposing nodes that can be labeled and repeats the interactions until the goal query is learned. More precisely, we analyze what it means for a node to be informative for the learning process, we show the intractability of deciding whether a node is informative or not, and we propose efficient strategies to present examples to the user.

- To evaluate our approach, we have run experiments on both real-world and synthetic datasets. Our study shows the effectiveness of the learning algorithm and the advantage of using an interactive strategy, which significantly reduces the number of examples needed to learn the goal query.

Finally, we would like to spend a few words on the class of queries that we investigate in this paper. As already mentioned, we focus on regular expressions, which are fundamental for graph query languages [7, 41] and lately used in the definition of SPARQL property paths[1]. Graph queries defined by regular expressions have been known as *regular path queries.* Intuitively, such queries retrieve pairs of nodes in the graph s.t. one can navigate between them with a path in the language of a given regular expression [7, 41]. Although the usual semantics of regular path queries is *binary* (i.e., selects pairs of nodes), in this paper we consider a generalization of this semantics that we call *monadic*, as it outputs only the originated nodes of the paths. The motivation behind using a monadic semantics is essentially threefold. First, it entails a larger space of potential solutions than a binary semantics. Indeed, with the latter semantics the end node of a path is fixed, which basically corresponds to have a smaller number of candidate paths that start at the originated node and that can be possibly labeled by the user. Second, in our learning framework, the amount of user effort should be kept as minimal as possible (which led to design an interactive scenario) and thus we let the user focus solely on the originated nodes of the paths rather than on pairs of nodes. Third, the development of the learning algorithm for monadic queries is extensible to binary queries and n-ary queries in a straightforward fashion, as shown in the paper.

Organization. In Section 2, we introduce some basic notions. In Section 3, we define our framework for learning from a set of examples, we present our learning algorithm, and we prove our learnability results. In Section 4, we propose an interactive algorithm and characterize the quantity of information of a node. In Section 5, we experimentally evaluate the performance of our algorithms. Finally, we conclude our paper and outline future directions in Section 6. Due to space restrictions, in this paper we omit the proofs of several results and we refer to the appendix of our technical report [11] for detailed proofs.

Related work

Learning queries from examples is a popular and interesting topic in databases. Very recently, algorithms for learning relational queries (e.g., quantifiers [1], joins [14, 15]) or XML queries (e.g., tree patterns [38]) have been proposed. Besides learning queries, researchers have investigated the learnability of relational schema mappings [40], as well as schemas [9] and transformations [30] for XML. A fairly close problem to learning is definability [4]. In this paragraph, we discuss the positioning of our own work w.r.t. these and other papers.

A wealth of research on using computational learning theory [26] has been recently conducted in databases [1, 9, 14, 30, 38, 40]. In this paper, we use *grammatical inference* [18] i.e., the branch of machine learning that aims at constructing a formal grammar by generalizing a set of examples. In particular, all the above papers on learning tree patterns [38], schemas [9, 16], and transformations [30] are based on it.

Our definition of learnability is inspired by the well-known framework of *language identification in the limit* [20], which requires a learning algorithm to be polynomial in the size of the input, *sound* (i.e., always return a concept consistent with the examples given by the user or a special *null* value if such concept does not exist) and *complete* (i.e., able to produce every concept with a sufficiently rich set of examples). In our case, we show that checking the consistency of a given set of examples is intractable, which implies that there is no algorithm able to answer *null* in polynomial time when the sample is inconsistent. This leads us to the conclusion that path queries are not learnable in the classical framework. Consequently, we slightly modify the framework and require the algorithm to learn the goal query in polynomial time if a polynomially-sized characteristic set of examples is provided. This learning framework has been recently employed for learning XML transformations [30] and is referred to as learning with *abstain* since the algorithm can abstain from answering when the characteristic set of examples is not provided.

The classical algorithm for learning a regular language from positive and negative word examples is RPNI [35], which basically works as follows: (i) construct a DFA (usually called prefix tree acceptor or PTA [18]) that selects all positive examples; (ii) generalize it by state merges while no negative example is covered. Unfortunately, RPNI cannot be directly adapted to our setting since the input positive and negative examples are not words in our case. Instead, we have a set of graph nodes starting from which we have to select (from a potentially infinite set) the paths that led the user to label them as examples. After selecting such paths, we generalize by state merges, similarly to RPNI.

A problem closely related to learning is *definability*, recently studied for graph databases [4]. Learning and definability have in common the fact that they look for a query consistent with a set of examples. The difference is that learning allows the query to select or not the nodes that are not explicitly labeled as positive examples while definability requires the query to select nothing else than the set of positive examples (i.e., all other nodes are implicitly negative). Nonetheless, some of the intractability proofs for definability can be adapted to our learning framework to show the intractability of consistency checking (cf. Section 3). To date, no polynomial algorithms have been yet proposed to construct path queries from a consistent set of examples.

2. GRAPH DATABASES AND QUERIES

In this section we define the concepts that we manipulate throughout the paper.

Alphabet and words. An *alphabet* Σ is a finite, ordered set of symbols. A *word* over Σ is a sequence $a_1 \dots a_n$ of symbols from Σ. By $|w|$ we denote the *length* of a word w. The *concatenation* of two words $w_1 = a_1 \dots a_n$ and $w_2 =$

[1]`http://www.w3.org/TR/sparql11-query/`

$b_1 \ldots b_m$, denoted $w_1 \cdot w_2$, is the word $a_1 \ldots a_n b_1 \ldots b_m$. By ε we denote the *empty word*. A *language* is a set of words. By Σ^* we denote the language of all words over Σ. We extend the order on Σ to the standard lexicographical order $\leqslant_{lex}$ on words over Σ and define a well-founded *canonical order* $\leqslant$ on words: $w \leqslant u$ iff $|w| < |u|$ or $|w| = |u|$ and $w \leqslant_{lex} u$.

Graph databases. A graph database is a finite, directed, edge-labeled graph [7, 41]. Formally, a *graph* (*database*) G over an alphabet Σ is a pair (V, E), where V is a set of *nodes* and $E \subseteq V \times \Sigma \times V$ is a set of *edges*. Each edge in G is a triple $(\nu_o, a, \nu_e) \in V \times \Sigma \times V$, where ν_o is the *origin* of the edge, ν_e is the *end* of the edge, and a is the *label* of the edge. We often abuse notation and write $\nu \in G$ and $(\nu_o, a, \nu_e) \in G$ instead of $\nu \in V$ and $(\nu_o, a, \nu_e) \in E$, respectively. For example, take in Figure 3 the graph G_0 containing 7 nodes and 15 edges over the alphabet $\{a, b, c\}$.

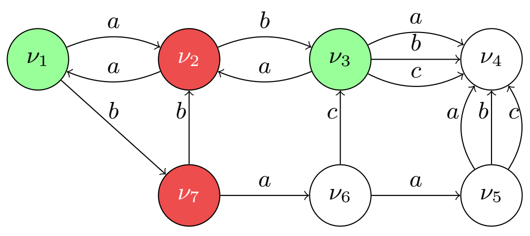


Figure 3: A graph database G_0.

Paths. A word $w = a_1 \ldots a_n$ *matches* a sequence of nodes $\nu_0 \nu_1 \ldots \nu_n$ if, for each $1 \leqslant i \leqslant n$, the triple (ν_{i-1}, a_i, ν_i) is an edge in G. For example, for the graph G_0 in Figure 3, the word aba matches the sequences of nodes $\nu_1\nu_2\nu_3\nu_4$ and $\nu_3\nu_2\nu_3\nu_4$, respectively, but does not match the sequence $\nu_1\nu_2\nu_7\nu_2$. Note that the empty word ε matches the sequence $\nu\nu$ for every $\nu \in G$. Given a node $\nu \in G$, by $paths_G(\nu)$ we denote the language of all words that match a sequence of nodes from G that starts by ν. In the sequel, we refer to such words as *paths*, and moreover, we say that a path w is *covered* by a node ν if $w \in paths_G(\nu)$. Paths are ordered using the canonical order $\leqslant$. For example, for the graph G_0 in Figure 3 we have $paths_{G_0}(\nu_5) = \{\varepsilon, a, b, c\}$. Note that $\varepsilon \in paths_G(\nu)$ for every $\nu \in G$. Moreover, note that $paths_G(\nu)$ is finite iff there is no cycle reachable from ν in G. For example, for the graph G_0 in Figure 3, $paths_{G_0}(\nu_1)$ is infinite. We naturally extend the notion of paths to a set of nodes i.e., given a set of nodes X from a graph G, by $paths_G(X) = \bigcup_{\nu \in X} paths_G(\nu)$.

Regular expressions and automata. A *regular language* is a language defined by a *regular expression* i.e., an expression of the following grammar:

$$q := \varepsilon \mid a \quad (a \in \Sigma) \mid q_1 + q_2 \mid q_1 \cdot q_2 \mid q^*,$$

where by "$\cdot$" we denote the *concatenation*, by "$+$" we denote the *disjunction*, and by "$*$" we denote the *Kleene star*. By $L(q)$ we denote the *language* of q, defined in the natural way [22]. For instance, the language of $(a \cdot b)^* \cdot c$ contains words like $c, abc, ababc$, etc. Regular languages can alternatively be represented by automata and we also refer to [22] for standard definitions of *nondeterministic finite word automaton* (*NFA*) and *deterministic finite word automaton* (*DFA*). In particular, we represent every regular language by its *canonical DFA* that is the unique smallest DFA that describe the language. For example, we present in Figure 4 the canonical DFA for $(a \cdot b)^* c$.

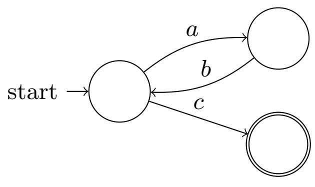


Figure 4: Canonical DFA for $(a \cdot b)^* c$.

Path queries. We focus on the class of path queries defined by regular expressions i.e., that select nodes having at least one *path* in the language of a given regular expression. Formally, given a graph G and a query q, we define the set of nodes *selected* by q on G:

$$q(G) = \{\nu \in G \mid L(q) \cap paths_G(\nu) \neq \varnothing\}.$$

For example, given the graph G_0 in Figure 3, the query a selects all nodes except ν_4, the query $(a \cdot b)^* \cdot c$ selects the nodes ν_1 and ν_3, and the query $b \cdot b \cdot c \cdot c$ selects no node. In the rest of the paper, we denote the set of all path queries by PQ and we refer to them simply as *queries*. We represent a query by its canonical DFA, hence the *size* of a query is the number of states in the canonical DFA of the corresponding regular language. For example, the size of the query $(a \cdot b)^* \cdot c$ is 3 (cf. Figure 4).

Equivalent queries. Two queries q and q' are *equivalent* if for every graph G they select exactly the same set of nodes i.e., $q(G) = q'(G)$. For example, the queries a and $a \cdot b^*$ are equivalent since each node having a path $ab \ldots b$ has also a path a. This example can be easily generalized and yields to defining the class of prefix-free queries. Formally, we say that a query q is *prefix-free* if for every word from $L(q)$, none of its prefixes belongs to $L(q)$. Given a query q, there exists a unique prefix-free query equivalent to q, which, moreover, can be constructed by simply removing all outgoing transitions of every final state in the canonical DFA of q. Our interest in prefix-free queries is that they can be seen as *minimal representatives* of *equivalence classes* of queries and as such they are desirable queries for learning. Indeed, every prefix-free query q is in fact equivalent to an infinite number of queries $q \cdot (q' + \varepsilon)$, where q' can be every PQ. In the remainder, we assume w.l.o.g. that all queries that we manipulate are prefix-free.

3. LEARNING FROM EXAMPLES

The input of a learning algorithm consists of a graph on which the user has annotated a few nodes as *positive* or *negative examples*, depending on whether or not she would like the nodes as part of the query result. Our goal is to exploit such examples to find a query that satisfies the user. In this paper, we explore two learning protocols: (i) the user provides a *sample* (i.e., a set of examples) that remains *fixed* during the learning process, and (ii) the learning algorithm *interactively* asks the user to label more examples until the learned query behaves exactly as the user wants.

First, we concentrate on the case of a fixed set of examples. We identify the challenges of such an approach, we show the unfeasability of the standard framework of *language identification in the limit* [20] and slightly modify it to propose a learning framework with *abstain* (Section 3.1). Next, we present a learning algorithm for the class of PQ (Section 3.2) and we identify the conditions that a graph and a sample must satisfy to allow polynomial learning of the user's goal query (Section 3.3). We study the case of query learning from user interactions in Section 4.

3.1 Learning framework

Given a graph $G = (V, E)$, an *example* is a pair (ν, α), where $\nu \in V$ and $\alpha \in \{+, -\}$. We say that an example of the form $(\nu, +)$ is a *positive example* while an example of the form $(\nu, -)$ is a *negative example*. A *sample* S is a set of examples i.e., a subset of $V \times \{+, -\}$. Given a sample S, we denote the set of positive examples $\{\nu \in V \mid (\nu, +) \in S\}$ by S_+ and the set of negative examples $\{\nu \in V \mid (\nu, -) \in S\}$ by S_-. A sample is *consistent* (with the class of PQ) if there exists a (PQ) query that selects all positive examples and none of the negative ones. Formally, given a graph G and a sample S, we say that S is *consistent* if there exists a query q s.t. $S_+ \subseteq q(G)$ and $S_- \cap q(G) = \emptyset$. In this case we say that q is *consistent with* S. For instance, take the graph G_0 in Figure 3 and the sample S s.t. $S_+ = \{\nu_1, \nu_3\}$ and $S_- = \{\nu_2, \nu_7\}$; S is consistent because there exist queries like $(a \cdot b)^* \cdot c$ or $c + (a \cdot b \cdot c)$ that are consistent with S.

Next, we want to formalize what means for a class of queries to be *learnable*, as it is usually done in the context of *grammatical inference* [18]. The standard learning framework is *language identification the limit* (*in polynomial time and data*) [20], which requires a learning algorithm to operate in time *polynomial* in the size of its input, to be *sound* (i.e., always return a query consistent with the examples given by the user or a special *null* value if no such query exists), and *complete* (i.e., able to produce every query with a sufficiently rich set of examples).

Since we aim at a polynomial time algorithm that returns a query consistent with a sample, we must first investigate the *consistency checking* problem i.e., deciding whether such a query exists. To this purpose, we first identify a necessary and sufficient condition for a sample to be consistent.

Lemma 3.1 *Given a graph G and a sample S, S is consistent iff for every $\nu \in S_+$ it holds that $paths_G(\nu) \nsubseteq paths_G(S_-)$.*

From this characterization we can derive that the fundamental problem of consistency checking is PSPACE-complete.

Lemma 3.2 *Given a graph G and a sample S, deciding whether S is consistent is PSPACE-complete.*

PROOF SKETCH. The membership to PSPACE follows from Lemma 3.1 and the known result that deciding the inclusion of NFAs is PSPACE-complete [39]. The PSPACE-hardness follows by reduction from the universality of the union problem for DFAs, known as being PSPACE-complete [28]. ▫

This implies that an algorithm able to always answer *null* in polynomial time when the sample is inconsistent does not exist, hence our class of queries is not learnable in the classical framework. One solution could be to study less expressive classes of queries. However, as shown by the following Lemma, consistency checking remains intractable even for a very restricted class of queries, referred as "SORE($\cdot$)" in [4].

Lemma 3.3 *Given a graph G and a sample S, deciding whether there exists a query of the form $a_1 \cdot \ldots \cdot a_n$ (pairwise distinct symbols) consistent with S is NP-complete.*

PROOF SKETCH. For the membership of the problem to NP, we point out that a non-deterministic Turing machine guesses a query q that is a concatenation of pairwise distinct symbols (hence of length bounded by $|\Sigma|$) and then checks whether q is consistent with S. The NP-hardness follows by reduction from 3SAT, well-known as being NP-complete. ▫

The proofs of Lemma 3.2 and 3.3 rely on techniques inspired by the definability problem for graph query languages [4]. We also point out that the same intractability results for consistency checking hold for binary semantics.

Another way to overcome the intractability of our class of queries is to relax the soundness condition and adopt a learning framework with *abstain*, similarly to what has been recently done for learning XML transformations [30]. More precisely, we allow the learning algorithm to answer a special value *null* whenever it cannot efficiently construct a consistent query. In practice, the *null* value is interpreted as "not enough examples have been provided". However, the learning algorithm should always return in polynomial time either a consistent query or *null*. As an additional clause, we require a learning algorithm to be *complete* i.e., when the input sample contains a *polynomially-sized characteristic sample* [18, 20], the algorithm must return the goal query. More formally, we have the following.

Definition 3.4 *A class of queries $\mathcal{Q}$ is* learnable with abstain in polynomial time and data *if there exists a polynomial learning algorithm learner that is:*

1. **Sound with abstain.** *For every graph G and sample S over G, the algorithm learner(G, S) returns either a query in $\mathcal{Q}$ that is consistent with S, or null if no such query exists or it cannot be constructed efficiently.*
2. **Complete.** *For every query $q \in \mathcal{Q}$, there exists a graph G and a polynomially-sized characteristic sample CS on G s.t. for every sample S extending CS consistently with q (i.e., $CS \subseteq S$ and q is consistent with S), the algorithm learner(G, S) returns q.*

Note that the polynomiality depends on the choice of a representation for queries and recall that we represent each PQ with its canonical DFA. Next, we present a polynomial learning algorithm fulfilling the two aforementioned conditions and we point out the construction of a polynomial characteristic sample to show the learnability of PQ.

3.2 Learning algorithm

In a nutshell, the idea behind our learning algorithm is the following: for each positive node, we seek the path that the user followed to label such a node, then we construct the disjunction of the paths obtained in the previous step, and we end by generalizing this disjunction while remaining consistent with both positive and negative examples.

More formally, the algorithm consists of two complementary steps that we describe next: *selecting the smallest consistent paths* and *generalizing* them.

Selecting the smallest consistent paths (SCPs). Since the labeled nodes in a graph may be the origin of multiple paths, classical algorithms for learning regular expressions from words, such as RPNI [35], are not directly applicable to our setting. Indeed, we have a set of nodes in the graph from which we have to first select (from a potentially infinite set) the paths responsible for their selection. Therefore, the first challenge of our algorithm is to select for each positive node a path that is not covered by any negative. We call such a path a *consistent path*. One can select consistent paths by simply enumerating (according to the canonical order $\leqslant$) the paths of each node labeled as positive and stopping when a consistent path for each node is found. We refer to the obtained set of paths as the set of *smallest consistent paths* (SCPs) because they are the smallest (w.r.t. $\leqslant$) consistent paths for each node. As an example, for the graph G_0 in Figure 3 and a sample s.t. $S_+ = \{\nu_1, \nu_3\}$ and $S_- = \{\nu_2, \nu_7\}$, we obtain the SCPs abc and c for ν_1 and ν_3, respectively. Notice that in this case the disjunction of the SCPs (i.e., the query $c + (a \cdot b \cdot c)$) is consistent with the input sample and one may think that a learning algorithm should return such a query. The shortcoming of such an approach is that the learned query would be always very simple in the sense that it uses only concatenation and disjunction. Since we want a learning algorithm that covers all the expressibility of PQ (in particular including the Kleene star), we need to extend the algorithm with a further step, namely the generalization. We detail such a step at the end of this section.

Another problem is that the user may provide an inconsistent sample, by labeling a positive node having no consistent path (cf. Lemma 3.1). To clarify when such a situation occurs, consider a simple graph such as the one in Figure 5 having one positive node (labeled with $+$) and two negative ones (labeled with $-$). We observe that the positive

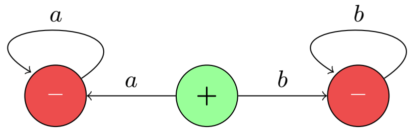


Figure 5: A graph with an inconsistent sample.

node has an infinite number of paths. However, all of them are covered by the two negative nodes, thus yielding an inconsistent sample. This example also shows that the simple procedure described above (i.e., enumerate the paths of the positive nodes and stop when finding consistent ones) fails because it leads to enumerating an infinite number on paths and never halting. On the other hand, we cannot perform consistency checking before selecting the SCPs since this fundamental problem is intractable (cf. Lemma 3.2 and 3.3). As a result, to avoid this possibly infinite enumeration, we choose to fix the maximal length of a SCP to a bound k. This bound engenders a new issue: for a fixed k it may not be possible to detect SCPs for all positive nodes. For instance, assume a graph that the user labels consistently with the query $(a \cdot b)^* \cdot c$, in particular she labels three positive nodes for which the SCPs are c, abc, and $ababc$. For a fixed $k = 3$, the SCP $ababc$ is not detected and the disjunction of the first two SCPs (i.e., $c + (a \cdot b \cdot c)$) is not a query consistent with the sample.

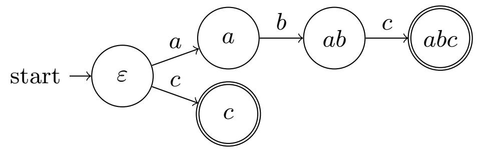
For all these reasons, we introduce a generalization phase in the algorithm, which permits to solve the two mentioned shortcomings i.e., (i) to learn a query that covers all the expressibility of PQ, and (ii) to select all positive examples even though not all of them have SCPs shorter than k.

Generalizing SCPs. We have seen how to select, whenever possible, a SCP of length bounded by k for each positive example. Next, we show how we can employ these SCPs to construct a more general query. The *learning algorithm* (Algorithm 1) takes as input a graph G and a sample S, and outputs a query q consistent with S whenever such query exists and can be built using SCPs of length bounded by k; otherwise, the algorithm outputs a special value *null*.

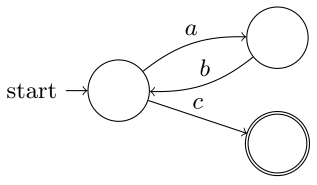
Algorithm 1 Learning algorithm – *learner*(G, S).

```
Input: graph $G$, sample $S$
Output: query $q$ consistent with $S$ or null
Parameter: fixed $k \in \mathbb{N}$ //maximal length of a SCP
1: for $\nu \in S_+ .\ \exists p \in \Sigma^{\leqslant k}.\ p \in paths_G(\nu) \setminus paths_G(S_-)$ do
2:     $P := P \cup \{\min_{\leqslant}(paths_G(\nu) \setminus paths_G(S_-))\}$
3: let $A$ be the prefix tree acceptor for $P$
4: while $\exists s, s' \in A.\ L(A_{s' \to s}) \cap paths_G(S_-) = \varnothing$ do
5:     $A := A_{s' \to s}$
6: if $\forall \nu \in S_+ .\ L(A) \cap paths_G(\nu) \neq \varnothing$ then
7:     return query $q$ represented by the DFA $A$
8: return null
```

We illustrate the algorithm on the graph G_0 in Figure 3 with a sample S s.t. $S_+ = \{\nu_1, \nu_3\}$ and $S_- = \{\nu_2, \nu_7\}$. For ease of exposition, we assume a fixed $k = 3$ (we explain how to obtain the value of k theoretically in Section 3.3 and empirically in Section 5). At first (lines 1-2), the algorithm constructs the set P of SCPs bounded by k for the positive nodes from which these paths in P can be constructed. Note that by $\Sigma^{\leqslant k}$ we denote the set of all paths of length at most k. For instance, on our example in Figure 3, we obtain $P = \{abc, c\}$. Then (line 3), we construct the *PTA* (*prefix tree acceptor*) [18] of P, which is basically a tree-like DFA accepting only the paths in P and having as states all their prefixes. Figure 6(a) illustrates the obtained PTA A for our example. Then (lines 4-5), we *generalize* A by merging two of its states if the obtained DFA selects no negative node. Note that by $A_{s' \to s}$ we denote the DFA obtained from A by modifying each occurrence of the state s' in s. Recall that on our example we have $P = \{abc, c\}$ and the PTA A in Figure 6(a). Next, we try to merge states of A: the states ε and a cannot be merged (because the obtained DFA would select the path bc that is covered by the negative ν_2), the states ε and c cannot be merged (because the path ε is covered by both negatives), while the states ε and ab can be merged without covering any negative example. On our example, we obtain the DFA in Figure 6(b), where no further states can be merged. Finally, the algorithm checks whether the query represented by A selects all the positive examples (not only those from whose SCPs we have constructed A), and if this is the case, it outputs the query (lines 6-7). In our case, the obtained $(a \cdot b)^* \cdot c$ selects all positive nodes hence is returned.



(a) Prefix tree acceptor.



(b) Result of generalization.

Figure 6: DFAs considered by *learner* for Figure 3.

3.3 Learnability results

Recall that in Section 3.1 we have formally defined what means for a class of queries to be learnable while in Section 3.2 we have proposed a learning algorithm for PQ. In this section, we prove that the proposed algorithm satisfies the conditions of Definition 3.4, thus showing the main theoretical result of the paper. We conclude the section with practical observations related to our learnability result.

By $\text{PQ}^{\leqslant n}$ we denote the PQ of size at most n.

Theorem 3.5 *The query class* $\text{PQ}^{\leqslant n}$ *is learnable with abstain in polynomial time and data, using the algorithm learner with the parameter k set to* $2 \times n + 1$.

PROOF. First, we point out that *learner* works in *polynomial time* in the size of the input. Indeed, the number of paths to explore for each positive node (lines 1-2) is polynomial since the length of paths is bounded by a fixed k. Then, both testing the consistency of queries considered during the generalization (lines 3-5) and testing whether the computed query selects all positive nodes (lines 6-7) reduce to deciding the emptiness of the intersection of two NFAs, known as being in PTIME [29]. Moreover, *learner* is *sound with abstain* since it returns either a query consistent with the input sample if it is possible to construct it using SCPs of length bounded by k, or *null* otherwise.

As for the *completeness* of *learner*, we show, for every $q \in \text{PQ}$, the construction of a graph G and of a characteristic sample CS on G with the properties from Definition 3.4 i.e., for every sample S that extends CS consistently with q (i.e., $CS \subseteq S$ and q is consistent with S), the algorithm $learner(G, S)$ returns q. The idea behind the construction is the following: we know that RPNI [35] is an algorithm that takes as input positive and negative word examples and outputs a DFA describing a consistent regular language, and moreover, the core of RPNI is based on DFA generalization via state merges similarly to *learner*; hence, for a query q, we want a graph and a sample on it s.t. when *learner* selects the SCPs, it should get exactly the words that RPNI needs for learning q if we see it as a regular language. Then, since RPNI is guaranteed to learn the goal regular language from these words, we infer that if *learner* selects and generalizes the SCPs corresponding to them, then *learner* is also guaranteed to learn the goal PQ.

We exemplify the construction on the query $q = (a \cdot b)^* \cdot c$. First, given q, we construct two sets of words P_+ and P_- that correspond to a characteristic sample used by RPNI to infer the regular language of q. In our case, we obtain $P_+ = \{c, abc\}$ and $P_- = \{\varepsilon, a, ab, ac, bc\}$. Then, the characteristic graph for learning the graph query q needs (i) for each $p \in P_+$, a node $\nu \in CS_+$ s.t. $p = \min_{\leqslant}(L(q) \cap paths_G(\nu))$, (ii) for each $p \in P_-$, a node $\nu \in CS_-$ s.t. $p \in paths_G(\nu)$, and (iii) for each p' that is smaller (w.r.t. the canonical order $\leqslant$) than a word $p \in P_+$ and is not prefixed by any word in $L(q)$, a node $\nu \in CS_-$ s.t. $p' \in paths_G(\nu)$. For our query $(a \cdot b)^* c$, (i) implies two positive nodes: a node ν s.t. $c \in paths_G(\nu)$ and another node ν' s.t. $abc \in paths_G(\nu')$ and $c \notin paths_G(\nu')$, while (ii) and (iii) imply a node ν'' s.t. $\nu'' \notin q(G)$ and $\{\varepsilon, a, ab, ac, bc\} \subseteq paths_G(\nu'')$ (cf. ii) and $\{\varepsilon, a, b, aa, ab, ac, ba, bb, bc, aaa, aab, aac, aba, abb\} \subseteq paths_G(\nu'')$ (cf. iii). In Figure 7 we illustrate such a graph and we highlight the two positive and one negative node examples.

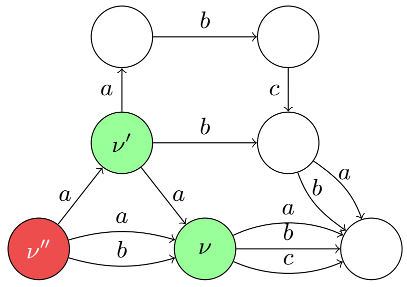


Figure 7: Graph from the proof of Theorem 3.5.

Recall that the size of a query is the number of states of its canonical DFA. According to the above construction, we need $|CS_+| = |P_+|$ and $|CS_-| = 1$. Since $|P_+|$ is polynomial in the size of the query [35], we infer that $|CS|$ is also polynomial in the size of the query. Moreover, to learn the regular language of a query of size n, the longest path in P_+ is of size $2 \times n + 1$ [35]. Hence, to be able to select this path with *learner* (assuming the presence of a characteristic sample), we need the parameter k of *learner* to be at least $2 \times n + 1$. Thus, for each possible size n of the goal query there exists a polynomial learning algorithm satisfying the conditions of Definition 3.4, which concludes the proof. □

We end this section with some practical observations related to our learnability result.

First, we point out that although Theorem 3.5 requires a certain theoretical value for k to guarantee learnability of queries of a certain size, our experiments indicate that small values of k (between 2 and 4) are enough to cover most practical cases. Then, even though the definition of learnability requires that one characteristic sample exists, in practice there may be many of such samples and there exists an infinite number of graphs on which we can build them. In fact, a graph that contains a subgraph with a characteristic sample is also characteristic.

Second, we also point out that a practical sample may be characteristic without having all negative paths on the same node as required by the aforementioned construction. For instance, the sample that we have used to illustrate the learning algorithm (i.e., the sample s.t. $S_+ = \{\nu_1, \nu_3\}$ and $S_- = \{\nu_2, \nu_7\}$ on the graph in Figure 3) is characteristic for $(a \cdot b)^* \cdot c$ and all above mentioned negatives paths are covered by two negative nodes.

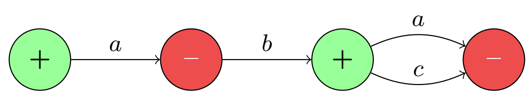


Figure 8: A graph and a sample for $(a \cdot b)^* \cdot c$.

Third, if a graph does not own a characteristic sample, the user's goal query on that graph cannot be exactly identified. In such a case, the learning algorithm returns a query equivalent to the goal query on the graph and hence *indistinguishable* by the user (i.e., they select exactly the same set of nodes). For instance, take the graph in Figure 8 and assume a user labeling the nodes w.r.t. the goal $(a \cdot b)^* \cdot c$. The learning algorithm returns the query a that selects exactly the same set of nodes on this graph.

Fourth, the presented learning techniques are directly applicable to different query semantics such as binary and n-ary queries. We give here only the intuition behind these applications and we point out that more details about the corresponding algorithms can be found in the appendix of our technical report [11]. To learn a binary query, the only change to Algorithm 1 is that each positive example implies a smaller number of candidate paths from which we have to choose a consistent one (since the destination node is also known). Then, for the n-ary case (i.e., when an example is a tuple of nodes labeled with $+$ or $-$), we have simply to apply the previous algorithm to learn a query for each position in the tuple and then to combine those.

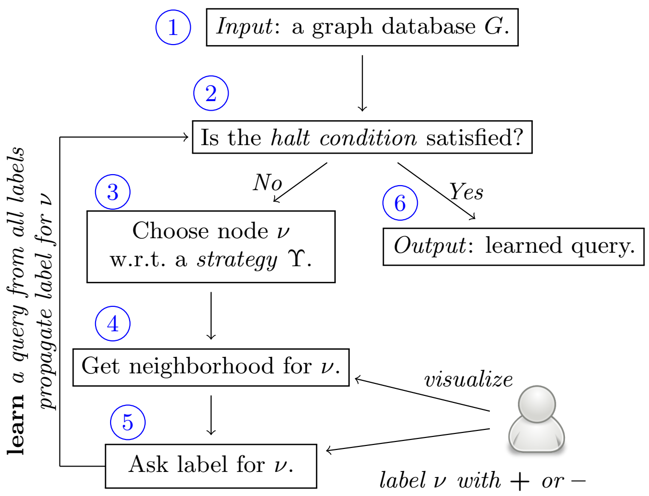
4. LEARNING FROM INTERACTIONS

In this section, we investigate the problem of query learning from a different perspective. In Section 3 we have studied the setting of a fixed set of examples provided by the user and no interaction with her during the learning process. In this section, we consider an *interactive scenario* where the learning algorithm starts with an empty sample and continuously interacts with the user and asks her to label additional nodes until she is satisfied with the output of the learned query. To this purpose, we first propose the *interactive scenario* (Section 4.1). Then, we discuss different parameters for it, in particular what means for a node to be *informative* and what is a *practical strategy* of proposing nodes to the user (Section 4.2). We also point out that we have employed the aforementioned interactive scenario as the core of a system for interactive path query specification on graphs [12].

4.1 Interactive scenario

We consider the following *interactive scenario*. The user is presented with a node of the graph and indicates whether the node is selected or not by the query that she has in mind. We repeat this process until a sufficient knowledge of the goal query has been accumulated (i.e., there exists at most one query consistent with the user's labels). This scenario is inspired by the well-known framework of *learning with membership queries* [3] and we have recently formalized it as a general paradigm for learning queries on big data [13]. In Figure 9, we describe the current instantiation for path queries on graphs and we detail next its different steps.

(1) (2) We consider as input a graph database G. Initially, we assume an empty sample that we enrich via simple interactions with the user. The interactions continue until



1
Input: a graph database G.
2
Is the *halt condition* satisfied?
No
Yes
3
Choose node ν w.r.t. a *strategy* Υ.
6
Output: learned query.
4
Get neighborhood for ν.
visualize
5
Ask label for ν.
label ν with $+$ or $-$
learn *a query from all labels*
propagate label for ν

Figure 9: Interactive scenario.

a *halt condition* is satisfied. A natural halt condition is to stop the interactions when there is exactly one consistent query with the current sample. In practice, we can imagine weaker conditions e.g., the user may stop the process earlier if she is satisfied by some candidate query proposed at some intermediary stage during the interactions.

(3) We propose nodes of the graph to the user according to a *strategy* Υ i.e., a function that takes as input a graph G and a sample S, and returns a node from G. Since our goal is to minimize the amount of effort needed to learn the user's goal query, a smart strategy should avoid proposing to the user nodes that do not bring any information to the learning process. The study of such strategies yields to defining the notion of *informative nodes* that we formalize in the next section.

(4) A node by itself does not carry enough information to allow the user to understand whether it is part of the query result or not. Therefore, we have to enhance the information of a node by zooming out on its *neighborhood* before actually showing it to the user. This step has the goal of producing a small, easy to visualize fragment of the initial graph, which permits the user to label the proposed node as a positive or a negative example. More concretely, in a practical scenario, all nodes situated at a distance k (as the parameter of Algorithm 1 explained in Section 3.2) should be sufficient for the user to decide whether she wants or not the proposed node. In any case, the user has neither to visualize all the graph that can be potentially large, nor to look by herself for interesting nodes because our interactive scenario proposes such nodes to the user.

(5) (6) The user visualizes the neighborhood of a given node ν and labels ν w.r.t. the goal query that she has in mind. Then, we propagate the given label in the rest of the graph and prune the nodes that become uninformative. Moreover, we run the learning algorithm *learner* (i.e., Algorithm 1 from Section 3.2), which outputs in polynomial time either a query consistent with all labels provided by the user, or *null* if such a query does not exist or cannot be constructed efficiently. When the halt condition is satisfied, we return the latest output of *learner* to the user. In particular, the halt condition may take into account such an

intermediary learned query q e.g., when the user is satisfied by the output of q on the instance and wants to stop the interactions.

In the next section, we precisely describe what means for a node to be informative for the learning process and what is a practical strategy of proposing nodes to the user.

4.2 Informative nodes and practical strategies

Before explaining the informative nodes, we first define the set of all queries consistent with a sample S over a graph G:

$$\mathcal{C}(G,S) = \{q \in \text{PQ} \mid S_+ \subseteq q(G) \wedge S_- \cap q(G) = \varnothing\}.$$

Assuming that the user labels the nodes consistently with some goal query q, the set $\mathcal{C}(G,S)$ always contains q. Initially, $S = \varnothing$ and $\mathcal{C}(G,S) = \text{PQ}$. Therefore, an ideal strategy of presenting nodes to the user is able to get us quickly from $S = \varnothing$ to a sample S s.t. $\mathcal{C}(G,S) = \{q\}$. In particular, a good strategy should not propose to the user the *certain nodes* i.e., nodes not yielding new information when labeled by the user. Formally, given a graph G, a sample S, and an unlabeled node $\nu \in G$, we say that ν is *certain* (w.r.t. S) if it belongs to one of the following sets:

$$\begin{aligned}
Cert_+(G,S) &= \{\nu \in G \mid \forall q \in \mathcal{C}(G,S).\ \nu \in q(G)\},\\
Cert_-(G,S) &= \{\nu \in G \mid \forall q \in \mathcal{C}(G,S).\ \nu \notin q(G)\}.
\end{aligned}$$

In other words, a node is certain with a label α if labeling it explicitly with α does not eliminate any query from $\mathcal{C}(G,S)$. For instance, take the graph in Figure 10 with a positive, a negative, and an unlabeled node, which belongs to $Cert_+$ because it is selected by the unique (prefix-free) query in $\mathcal{C}(G,S)$ (i.e., b). Additionally, we observe that labeling it otherwise (i.e., with a $-$) leads to an inconsistent sample. The notion of certain nodes is inspired by possible world semantics and certain answers [23], and already employed for XML querying for non-expert users [17] and for the inference of relational joins [14].

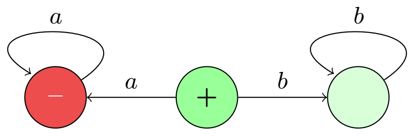


Figure 10: Two labeled nodes and a certain node.

Next, we give necessary and sufficient conditions for a node to be certain, for both positive and negative labels:

Lemma 4.1 *Given a sample S and a node ν from G:*

1. *$\nu \in Cert_+(G,S)$ iff there exists $\nu' \in S_+$ s.t. $paths_G(\nu') \subseteq paths_G(S_-) \cup paths_G(\nu)$,*
2. *$\nu \in Cert_-(G,S)$ iff $paths_G(\nu) \subseteq paths_G(S_-)$.*

Additionally, given a graph G, a sample S, and a node ν, we say that ν is *informative* (w.r.t. S) if ν has not been labeled by the user nor it is certain. Unfortunately, by using the characterization from Lemma 4.1, we derive the following.

Lemma 4.2 *Given a graph G and a sample S, deciding whether a node ν is informative is PSPACE-complete.*

An intelligent strategy should propose to the user only informative nodes. Since deciding the informativeness of a node is intractable (cf. Lemma 4.2), we need to explore *practical strategies* that efficiently compute the next node to label. Consequently, we propose two simple but effective strategies that we detail next and that we have evaluated experimentally. The basic idea behind them is to avoid the intractability of deciding informativeness of a node by looking only at a small number of paths of that node. More precisely, we say that a node is *k-informative* if it has at least one path of length at most k that is not covered by a negative example. If a node is k-informative, then it is also informative, otherwise we are not able to establish its informativeness w.r.t. the current k. Then, strategy kR consists of taking *randomly* a k-informative node while strategy kS consists of taking the k-informative node having the *smallest* number of non-covered k-paths, thus favoring the nodes for which computing the SCPs is easier. In the next section, we discuss the performance of these strategies as well as how we set the k in practice.

5. EXPERIMENTS

In this section, we present an experimental study devoted to gauge the performance of our learning algorithms. In Section 5.1, we introduce the used datasets: the *AliBaba biological graph* and randomly generated *synthetic graphs*. In Section 5.2 and Section 5.3, we present the results for the two settings under study: *static* and *interactive*, respectively. Our algorithms have been implemented in C and our experiments have been run on an Intel Core i7 with 4 × 2.9 GHz CPU and 8 GB RAM.

5.1 Datasets

Despite the increasing popularity of graph databases, benchmarks allowing to assess the performance of emerging graph database applications are still lacking or under construction [10]. In particular, there is no established benchmark devoted to graph queries defined by regular expressions. Due to this lack, we have adopted a real dataset recently used by [27] to evaluate the performance of optimization algorithms for regular path queries. This dataset, called *AliBaba* [36], represents a real graph from research on biology, extracted by text mining on PubMed. The dataset has a semantic part consisting of a network of protein-protein interactions and a textual part, reporting text co-occurrence for words. The first part was more appropriate to apply our learning algorithms than the second. Therefore, we have extracted the semantic part from the original graph, thus obtaining a subgraph of about 3k nodes and 8k edges. Similarly, from the set of real-life queries reported in [27], we have retained those that select at least one node on the graph to obtain at least one positive example for learning. Thus, we have used 6 biological queries (denoted by $bio_1, \ldots, bio_6$), which are structurally complex and have selectivities varying from 1 to a total of 711 nodes i.e., from 0.03% to 22% of the nodes of the graph. We summarize these queries in Table 1. By small letters a, b we denote symbols from the alphabet while by capital letters A, C, E, I we denote disjunctions of symbols from the alphabet i.e., expression of the form $a_1 + \ldots + a_n$. These disjunctions contain up to 10 symbols, with possibly overlapping ones among them.

Additionally, we have implemented a synthetic data generator, which yields graphs of varying size and similar to real-world graphs. The latest feature let us generate *scale-free* graphs with a *Zipfian* edge label distribution [27]. We report here the results for generated graphs of size 10k, 20k,

	Query	*Selectivity*
bio_1	$b \cdot A \cdot A^*$	0.03%
bio_2	$C \cdot C^* \cdot a \cdot A \cdot A^*$	0.2%
bio_3	$C \cdot E$	3%
bio_4	$I \cdot I \cdot I^*$	11%
bio_5	$A \cdot A \cdot A^* \cdot I \cdot I \cdot I^*$	12%
bio_6	$A \cdot A \cdot A^*$	22%

Table 1: Biological queries.

and 30k nodes, and with a number of edges three times larger. Moreover, we focus on synthetic queries that are similar in structure to the aforementioned real-life biological queries. In particular, the three queries that we report here (denoted by syn_1, syn_2, and syn_3) have the structure $A \cdot B^* \cdot C$, where A, B, and C are disjunctions of up to 10 symbols, with overlapping ones among them. The difference between these three queries is w.r.t. their selectivity: regardless the actual size of the graph, syn_1, syn_2, and syn_3 select 1%, 15%, and 40% of the graph nodes, respectively.

Before presenting the experimental results, we say a few words about how we set empirically the parameter k from *learner*. Since in our experiments we assume that the user labels the nodes of the graph consistently with some goal query, the input sample is always consistent. Hence, there exists a consistent path for each positive node and we dynamically discover the length of the SCPs. In particular, we start with $k = 2$; if for a given k, the query learned using SCPs shorter that k does not select all positive nodes, we increment k and iterate. For the interactive case, the aforementioned procedure becomes: start with $k = 2$; seek k-informative nodes (cf. Section 4.2) and increase k when the current k does not yield any k-informative node. In practice, in the majority of cases $k = 2$ is sufficient and it may reach values up to 4 in some isolated cases.

5.2 Static experiments

The setup of static experiments is as follows. Given a graph and a goal query, we take as positive examples some random nodes of the graph that are selected by the query and as negative examples some random nodes that are not selected by it. All these examples are given as input to *learner*, which returns a consistent query. We consider the learned query as a binary classifier and we measure the F1 score w.r.t. the goal query. Thus, for different percentages of labeled nodes in the graph, we measure the F1 score of the learned query along with the learning time. We present the summary of results in Figure 11 and 12, which show the F1 score and the learning time for the biological (a) and synthetic queries (b, c, d), respectively. Since the positive effect of the generalization in addition to the selection of SCPs is generally of 1% in F1 score, we do not highlight the two steps of the algorithm for the sake of figure readability.

We can observe that, not surprisingly, by increasing the percentage of labeled nodes of the graph, the F1 score also increases (Figure 11). Overall, the F1 score is 1 or sufficiently close to 1 for all queries, except a few cases that we discuss below. The worst behavior is clearly exhibited by query bio_5, when we can observe that the F1 score converges to 1 less faster than the others. For this query, we can also observe that the learning time is higher (Figure 12(a)). This is due to the fact that the graph is not characteristic for it (cf. Section 3.3), hence the selection of SCPs yields paths that are not relevant for the target query.

For what concerns the learning time, this remains reasonable (of the order of seconds) for all the queries and for both datasets. The most selective queries (bio_4, bio_5, bio_6) are more problematic since they entail a larger number of positive nodes in the step of selection of the SCPs. As a conclusion, notice that even when the F1 score is very high, these results on the static scenario are not fully beneficial since we need to label at least 7% of the graph nodes to have an F1 score equal to 1. As we later show with the interactive experiments, we can significantly reduce the number of labels needed to reach an F1 score equal to 1.

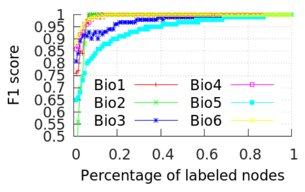
Finally, the synthetic experiments on various graph sizes and query selectivities confirm the aforementioned observations. In particular, when the queries are more selective (as syn_2 and syn_3), increasing the number of examples implies more visible changes in the learning time (cf. Figure 12). Additionally, we observe the goal queries with higher selectivity converge faster to a F1 score equal to 1 (cf. Figure 11). Intuitively, this is due to the fact that such cases imply a bigger number of positive examples from which the learning algorithm can benefit to generalize faster the goal query.

5.3 Interactive experiments

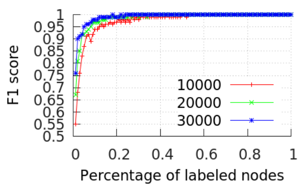
The setup of interactive experiments is as follows. Given a graph and a goal query, we start with an empty sample and we continuously select a node that we ask the user to label, until the learned query selects exactly the same set of nodes as the goal query or, in other words, until the goal query and the learned query are indistinguishable by the user (cf. Section 3.3). This corresponds to obtaining an F1 score of 1. In this setting, we measure the percentage of the labeled nodes of the graph and the learning time i.e., the time needed to compute the next node to label. In particular, the number of interactions corresponds to the total number of examples, the latter being the sum of the number of positive examples and the number of negative examples. The summary of interactive experiments is presented in Table 2.

We can observe that, differently from the static scenario, labeling around 1% of the nodes of the graph suffices to learn a query with F1 score equal to 1. Even for the most difficult one (bio_5), we get a rather significant improvement, as the percentage of interactions is drastically reduced to 7.7% of the nodes in the graph (while in the static case it was 87% of the nodes in the graph). Overall, these numbers prove that the interactive scenario considerably reduces the number of examples needed to infer a query of F1 score equal to 1. The synthetic experiments confirm this behavior for various graph sizes and query selectivities. While learning a query with F1 score equal to 1 corresponds to the strongest halt condition of our interactive scenario (cf. Figure 9), we believe that in practice the user may choose to stop the interactions earlier (and hence label less nodes) if she is satisfied by an intermediate query proposed by the learning algorithm. We consider such halt conditions in our system demo [12].

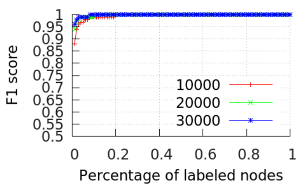
Moreover, these experiments also show that the two strategies (kS and kR, cf. Section 4.2) have a similar behavior, even though kS is slightly better (w.r.t. minimizing the number of interactions) for the most selective queries. Intuitively, this happens because such a strategy favors the nodes for which computing the SCPs has a smaller space of solutions (cf. Section 4.2). Finally, we can observe that the two strategies are also efficient, as they lead to a learning time of the order of seconds in all cases.



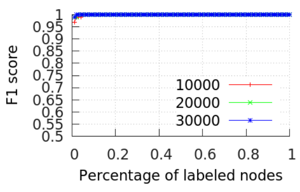
(a) Biological queries.



(b) Synthetic query syn_1.

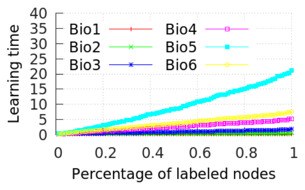


(c) Synthetic query syn_2.

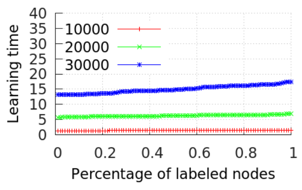


(d) Synthetic query syn_3.

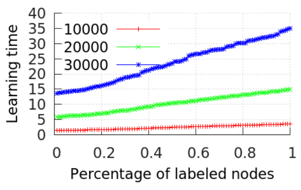
Figure 11: Summary of static experiments – F1 score.



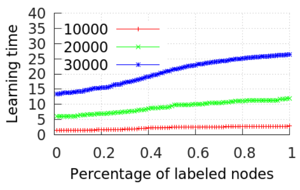
(a) Biological queries.



(b) Synthetic query syn_1.



(c) Synthetic query syn_2.



(d) Synthetic query syn_3.

Figure 12: Summary of static experiments – Learning time (seconds).

Dataset	*Bio query / Graph size*	*Labels needed for F1 score = 1* **without interactions**	*Interactive strategy*	*Labels needed for F1 score = 1* **with interactions**	*Time between interactions (seconds)*
Biological queries	bio_1	7%	kR	0.06%	0.19
			kS	0.06%	0.33
	bio_2	7%	kR	1.78%	0.26
			kS	3.13%	0.48
	bio_3	66%	kR	1.24%	0.34
			kS	1.49%	0.45
	bio_4	12%	kR	1.32%	0.23
			kS	0.22%	0.53
	bio_5	87%	kR	7.7%	3.45
			kS	7.39%	3.79
	bio_6	12%	kR	1.18%	0.24
			kS	0.35%	0.3
Synthetic query syn_1	10000	51%	kR	0.15%	1.33
			kS	0.17%	1.35
	20000	26%	kR	0.07%	5.83
			kS	0.06%	5.92
	30000	22%	kR	0.04%	13.5
			kS	0.04%	13.95
Synthetic query syn_2	10000	20%	kR	0.38%	1.57
			kS	0.36%	1.58
	20000	11%	kR	0.23%	6.63
			kS	0.22%	6.78
	30000	8%	kR	0.17%	15.24
			kS	0.16%	15.38
Synthetic query syn_3	10000	5%	kR	0.1%	1.32
			kS	0.1%	1.32
	20000	3%	kR	0.05%	5.66
			kS	0.05%	5.68
	30000	2%	kR	0.04%	13.15
			kS	0.04%	13.41

Table 2: Summary of interactive experiments.

6. CONCLUSIONS AND FUTURE WORK

We have studied the problem of learning path queries defined by regular expressions from user examples. We have identified fundamental difficulties of the problem setting, formalized what means for a class of queries to be learnable, and shown that the above class enjoys learnability. Additionally, we have investigated an interactive scenario, analyzed what means for a node to be informative, and proposed practical strategies of presenting examples to the user. Finally, we have shown the effectiveness of the algorithms and the improvements of using an interactive approach through an experimental study on both real and synthetic datasets.

We envision several directions of our work, one of which being to sample a graph and finding informative nodes on representative samples, in the spirit of [31]. Moreover, motivated by the absence of benchmarks devoted to queries defined by regular expressions, we want to develop such a benchmark. This would permit to better analyze the performance of algorithms involving regular expressions on graphs, including the learning algorithms proposed in this paper.

Acknowledgements. We would like to thank the authors of [27] for sharing the AliBaba dataset and to Sarah Cohen-Boulakia for her comments on a draft of the paper.

7. REFERENCES

[1] A. Abouzied, D. Angluin, C. H. Papadimitriou, J. M. Hellerstein, and A. Silberschatz. Learning and verifying quantified boolean queries by example. In *PODS*, pages 49–60, 2013.

[2] R. Angles and C. Gutiérrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1), 2008.

[3] D. Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1988.

[4] T. Antonopoulos, F. Neven, and F. Servais. Definability problems for graph query languages. In *ICDT*, pages 141–152, 2013.

[5] M. Arenas and J. Pérez. Querying semantic web data with SPARQL. In *PODS*, pages 305–316, 2011.

[6] G. Bagan, A. Bonifati, and B. Groz. A trichotomy for regular simple path queries on graphs. In *PODS*, pages 261–272, 2013.

[7] P. Barceló. Querying graph databases. In *PODS*, pages 175–188, 2013.

[8] K. Belhajjame. Annotating the behavior of scientific modules using data examples: A practical approach. In *EDBT*, pages 726–737, 2014.

[9] G. J. Bex, W. Gelade, F. Neven, and S. Vansummeren. Learning deterministic regular expressions for the inference of schemas from XML data. *TWEB*, 4(4), 2010.

[10] P. A. Boncz, I. Fundulaki, A. Gubichev, J.-L. Larriba-Pey, and T. Neumann. The linked data benchmark council project. *Datenbank-Spektrum*, 13(2):121–129, 2013.

[11] A. Bonifati, R. Ciucanu, and A. Lemay. Learning path queries on graph databases, 2014. TR at `http://hal.inria.fr/hal-01068055`.

[12] A. Bonifati, R. Ciucanu, and A. Lemay. Interactive path query specification on graph databases. In *EDBT*, 2015.

[13] A. Bonifati, R. Ciucanu, A. Lemay, and S. Staworko. A paradigm for learning queries on big data. In *Data4U*, pages 7–12, 2014.

[14] A. Bonifati, R. Ciucanu, and S. Staworko. Interactive inference of join queries. In *EDBT*, pages 451–462, 2014.

[15] A. Bonifati, R. Ciucanu, and S. Staworko. Interactive join query inference with JIM. *PVLDB*, 7(13):1541–1544, 2014.

[16] R. Ciucanu and S. Staworko. Learning schemas for unordered XML. In *DBPL*, pages 31–40, 2013.

[17] S. Cohen and Y. Weiss. Certain and possible XPath answers. In *ICDT*, pages 237–248, 2013.

[18] C. de la Higuera. *Grammatical Inference: Learning Automata and Grammars.* Cambridge University Press, 2010.

[19] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. CrowdDB: answering queries with crowdsourcing. In *SIGMOD Conference*, pages 61–72, 2011.

[20] E. M. Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, 1978.

[21] C. Heinlein. Workflow and process synchronization with interaction expressions and graphs. In *ICDE*, pages 243–252, 2001.

[22] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[23] T. Imielinski and W. Lipski Jr. Incomplete information in relational databases. *J. ACM*, 31(4):761–791, 1984.

[24] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu. Making database systems usable. In *SIGMOD Conference*, pages 13–24, 2007.

[25] N. Jayaram, A. Khan, C. Li, X. Yan, and R. Elmasri. Towards a query-by-example system for knowledge graphs. In *GRADES*, 2014.

[26] M. J. Kearns and U. V. Vazirani. *An introduction to computational learning theory.* MIT Press, 1994.

[27] A. Koschmieder and U. Leser. Regular path queries on large graphs. In *SSDBM*, pages 177–194, 2012.

[28] D. Kozen. Lower bounds for natural proof systems. In *FOCS*, pages 254–266, 1977.

[29] K.-J. Lange and P. Rossmanith. The emptiness problem for intersections of regular languages. In *MFCS*, pages 346–354, 1992.

[30] G. Laurence, A. Lemay, J. Niehren, S. Staworko, and M. Tommasi. Learning sequential tree-to-word transducers. In *LATA*, pages 490–502, 2014.

[31] J. Leskovec and C. Faloutsos. Sampling from large graphs. In *KDD*, pages 631–636, 2006.

[32] K. Losemann and W. Martens. The complexity of regular expressions and property paths in SPARQL. *ACM Trans. Database Syst.*, 38(4):24, 2013.

[33] D. Mottin, M. Lissandrini, Y. Velegrakis, and T. Palpanas. Exemplar queries: Give me an example of what you need. *PVLDB*, 7(5):365–376, 2014.

[34] A. Nandi and H. V. Jagadish. Guided interaction: Rethinking the query-result paradigm. *PVLDB*, 4(12):1466–1469, 2011.

[35] J. Oncina and P. García. Inferring regular languages in polynomial update time. *Pattern Recognition and Image Analysis*, pages 49–61, 1992.

[36] P. Palaga, L. Nguyen, U. Leser, and J. Hakenberg. High-performance information extraction with AliBaba. In *EDBT*, pages 1140–1143, 2009.

[37] R. Ronen and O. Shmueli. SoQL: A language for querying and creating data in social networks. In *ICDE*, pages 1595–1602, 2009.

[38] S. Staworko and P. Wieczorek. Learning twig and path queries. In *ICDT*, pages 140–154, 2012.

[39] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time: Preliminary report. In *STOC*, pages 1–9, 1973.

[40] B. ten Cate, V. Dalmau, and P. G. Kolaitis. Learning schema mappings. *ACM Trans. Database Syst.*, 38(4):28, 2013.

[41] P. T. Wood. Query languages for graph databases. *SIGMOD Record*, 41(1):50–60, 2012.

[42] L.-L. Yan, R. J. Miller, L. M. Haas, and R. Fagin. Data-driven understanding and refinement of schema mappings. In *SIGMOD Conference*, pages 485–496, 2001.

[43] M. M. Zloof. Query by example. In *AFIPS National Computer Conference*, pages 431–438, 1975.

TimeReach: Historical Reachability Queries on Evolving Graphs

Konstantinos Semertzidis, Kostas Lillis, Evaggelia Pitoura
Computer Science and Engineering Department
University of Ioannina, Greece
{ksemer,klillis,pitoura}@cs.uoi.gr

ABSTRACT

Since most graphs evolve over time, it is useful to be able to query their history. We consider historical reachability queries that ask for the existence of a path in some time interval in the past, either in the whole duration of the interval (conjunctive queries), or in at least one time instant in the interval (disjunctive queries). We study both alternatives of storing the full transitive closure of the evolving graph and of performing an online traversal. Then, we propose an appropriate reachability index, termed TimeReach index, that exploits the fact that most real-world graphs contain large strongly connected components. Finally, we present an experimental evaluation of all approaches, for different graph sizes, historical query types and time granularities.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems query processing

General Terms

Algorithms, Measurement, Performance

Keywords

Evolving Graphs, Historical Queries, Reachability

1. INTRODUCTION

In recent years, increasing amounts of graph structured data are being made available from a variety of sources, such as social, citation, computer and hyperlink networks. Almost all such real-world networks evolve over time, as nodes and edges are added or deleted. Analysis of their evolution finds a large spectrum of applications, ranging from social network marketing, to virus propagation and digital forensics.

In this paper, we assume that we are given an evolving set of graph snapshots corresponding to the state of the graph at different time instants. We address the problem of efficiently answering queries that involve such snapshots. In particular, we focus on a basic query type, namely reachability queries, that ask whether a node u was reachable from another node v during specific time intervals in the past. We call such queries *historical reachability queries*.

Although, there has been considerable interest in processing graph data, through a variety of graph queries including reachability, distance and pattern-based ones, querying the graph history is much less studied. The only other two approaches to building indexes for processing historical graph queries that we are aware of consider historical shortest-path queries [9, 2]. Specifically, the authors of [9] propose a method based on ordering nodes or edges pertinent to shortest path computation, while the dynamic index construction proposed in [2] does not support node or edge deletions.

All other work on historical queries focuses mainly on efficiently storing and retrieving the graph snapshots required for processing each query [14, 13, 21, 17]. In particular, in [14], a combination of graph deltas and selected materialized snapshots are explored, while in [13], the focus is on storing, sharing and processing deltas. In [21], temporally close snapshots are clustered, one representative per cluster is selected and used for an initial evaluation of the query. Finally, in [17], the placement and replication of snapshots in a distributed setting is studied. Instead, in this paper, we address the problem of building indexes for answering historical reachability queries.

Reachability queries on static graphs have been extensively studied. Research in this area follows two general directions through efficiently storing the transitive closure and speeding-up online traversal. With regards to transitive closure, various approaches have been proposed including the chain method [10, 5], methods exploring spanning trees, bit-vector compression [26] and interval [1, 28, 12], and hop [7, 22, 6] labeling. In the case of online traversal, often interval labeling [4, 25, 30] is used to prune the search space. There has also been some work on incrementally maintaining the reachability indexes in case of evolving graphs [1, 3, 23, 31], however, reachability still considers a single snapshot, i.e., the current version of the graph.

In this paper, we explore a compact representation of graph snapshots, called *version graph*, where each node and edge is annotated with the set of time intervals during which the corresponding node and edge existed in the evolving graph. We call such sets *lifespans* and seek for their minimum representation through using non-overlapping and non-continuous intervals. We also introduce a set of basic operations for efficiently manipulating lifespans of paths.

For processing historical reachability queries, we start by revisiting the basic transitive closure and online traversal approaches. For the transitive closure, we compute a minimum representation of reachability information for each pair of nodes. For the online traversal, we propose a novel interval-based traversal of the version graph along with a number of pruning steps. Furthermore, to avoid the cost and space overheads associated with precomputing the transitive closure and improving the processing cost of the online traversal, we propose a new approach, termed *TimeReach*.

TimeReach exploits the fact that most graphs consist of strongly connected components (SCCs) [20, 15]. Thus, instead of maintaining reachability information for pairs of nodes, we maintain *posting lists* with information about node membership in SCCs. We minimize the size of posting lists through an appropriate assignment of identifiers to SCCs. We show that the problem of the optimal assignment of identifiers to SCCs is equivalent to the maximum bipartite matching problem among SCCs in consequent graph snapshots. Along with postings, we maintain a condensed version graph which corresponds to the version graph of the SCCs evolution. To improve the performance of answering historical queries, we also introduce an interval-2hop approach based on pruned landmark labeling [2, 29] on the condensed version graph.

We have extensively evaluated our approach with three real social network datasets. Our experimental results show that TimeReach is space efficient, in particular for graphs consisting of large SCCs as is the case of social networks. Its incremental construction is fast; indexing a new snapshot graph takes just a few seconds. Finally, processing historical queries using TimeReach is orders of magnitude faster than the online traversal of the version graph.

The rest of this paper is structured as follows. In Section 2, we present related work, while in Section 3, we formally define historical reachability queries. In Section 4, we introduce the version graph and operations on lifespans and present the two baseline approaches, namely, the transitive closure and online traversal. In Section 5, we introduce the TimeReach Index approach, while in Section 6, we present experimental results. Section 7 concludes the paper.

2. RELATED WORK

Although, graph data management has been the focus of much current research, work in processing historical queries is rather limited. The main focus of research on evolving graphs has been on efficiently storing and retrieving graph snapshots. In this paper, our focus in on indexing for processing queries. To this end, we assume a compact representation of the sequence of graph snapshots in the form of a version graph. Alternatively, one can store just some subset of the graph snapshots in the sequence along with appropriate deltas, such that, any other snapshot can be reconstructed by applying the deltas on the selected snapshots [14, 13]. Various optimizations for reducing the storage and snapshot re-construction overheads have been proposed, such as a hierarchical index of deltas and a memory pool for the overlapping storage of snapshots [13]. Clustering temporally close snapshots and computing a representative for each cluster was also proposed [21], Deltas from representatives are stored for each cluster to achieve high compression. In the G* graph database, snapshots are efficiently stored by taking advantage of commonalities among them [16]. Different versions of each node are stored only once regardless of the number of snapshots it belongs to, and indexed by a compact in-memory index. For load balance and availability snapshot data are replicated among a number of workers.

Historical query processing in these approaches requires as a first and costly step reconstructing the relevant snapshots. Then, queries are processed through an online traversal on each of them. Query performance is addressed by trying to minimize the number of snapshots that need to be reconstructed by minimizing the number of deltas applied [14, 13], avoiding the reconstruction of all snapshots [21], or by parallel query execution and proper snapshot placement and distribution [17]. In this work, we address a different problem, that of indexing for historical reachability queries.

Historical shortest path distance queries were addressed in [9]. The authors propose a method based on ordering nodes or edges pertinent to shortest path computation. Finally, the recent work of [2] also proposes a dynamic indexing scheme for historical distance queries. However, the authors consider only insertions. This assumption simplifies the problem, since two nodes that are reachable remain reachable. The authors propose a dynamic 2hop index construction that is not applicable in the case of node or edge deletions.

Reachability queries on static graphs have been thoroughly investigated along two general directions: transitive closure compression and improving online search.

Transitive Closure Compression. Related research aims at compressing the transitive closure by storing for each node only a subset of the nodes it can reach. The first idea is to decompose the graph in k node-disjoint chains and for each node store only the first node it can reach in each chain [10, 5]. Another line of research extracts a spanning tree of the graph, and uses it to compress the transitive closure. Each node of the tree is labeled with an interval of integers such that if node u is an ancestor of v, the interval of u contains that of v. Reachability through tree edges can be easily determined by a label containment check. To incorporate reachability through non-tree edges each node inherits the intervals of its successors in the graph [1], or a partial transitive closure of non-tree edges is constructed [28]. Building upon the idea of interval labeling, a tree whose vertices are pair-wise disjoint paths extracted from the original graph is used in [12]. Another approach in compressing the transitive closure is 2-hop labeling [7, 22, 6]. Each node stores two sets of intermediate nodes: a set L_{out} of nodes it can reach and a set L_{in} of nodes that can reach it. Node u can reach node v only if $L_{out}(u) \cap L_{in}(v) \neq \emptyset$.

Speeding-up Online Traversal. These methods use interval labeling to aid online traversal by pruning the search space. In [4] and [25], a tree cover of the graph is constructed and then, for the queries that can not be answered by the tree labeling, an online search on the non-tree edges is performed using the labeling to guide the search. In [30], multiple intervals are used for the labeling. If the label containment check does not produce a negative answer, the graph is traversed online using the intervals for pruning the search.

Some of the works discuss the incremental maintenance of the index in the case of evolving graphs [1, 3, 23, 31]. However, the updated index contains reachability information only about the current version of the graph and cannot be used for answering historical queries.

The presented approaches are orthogonal to our approach in that they can be adapted so that they can be used to speed-up or avoid the online traversal of the condensed graph. We have demonstrated this by adapting, one of them, namely 2hop labeling.

3. PROBLEM DEFINITION

Most real world graphs evolve over time as new nodes or edges are added, or existing nodes or edges are deleted. We assume that time is discrete and use successive integers to denote successive time instants. There are two intuitive interpretations of time instants. One interpretation is that of actual time, for example time instant t may correspond to say October 20, 2014, 5:00am PDT. Another view is operational. In this case, time is advanced each time a graph operation, update or delete, occurs. Both interpretations of time instants are consistent with our representation.

Let $G = (V, E)$ be a directed graph where V is the set of nodes and E the set of edges. We use $G_t = (V_t, E_t)$ to denote the *graph snapshot* at time instant t, that is, the set of nodes and edges that exist at time instant t.

Definition 1 (evolving graph). *An evolving graph $\mathcal{G}_{[t_i,t_j]}$ in time interval $[t_i, t_j]$ is a sequence $\{G_{t_i}, G_{t_i+1}, \ldots, G_{t_j}\}$ of graph snapshots.*

An example is shown in Figure 1(a) which depicts an evolving graph $\mathcal{G}_{[t_0,t_3]}$ consisting of four graph snapshots $\{G_{t_0}, G_{t_1}, G_{t_2}, G_{t_3}\}$. For brevity, we denote time instant $t_i + 1$ as t_{i+1} and use t_i and i interchangeably, when the meaning is clear from context.

We use the term *time granularity* to refer to how often a new time instant and the corresponding graph snapshot are created. In the case of actual time, granularity may range for example from milliseconds to years, whereas, in the case of operational time, granularity may be at the level of one or more operations. A fine-grained time granularity necessitates maintaining a large amount of historical information, but supports precise historical queries.

Given a static directed graph $G = (V, E)$ and two nodes $u, v \in V$, a *reachability query* asks whether there exists a path from u to v in G. For evolving graphs, we introduce the following two types of historical reachability queries.

Definition 2 (historical reachability query). *Let $\mathcal{G}_{[t_i,t_j]} = \{G_{t_i}, G_{t_i+1}, \ldots G_{t_j}\}$, be an evolving graph, $I_Q = [t_k, t_l] \subseteq [t_i, t_j]$ a time interval and v, u a pair of nodes:*

(i) a conjunctive historical reachability query $u \overset{I_{Q\wedge}}{\rightsquigarrow} v$ returns true, if there exists a path from u to v in all graph snapshots G_{t_m}, $t_k \leq t_m \leq t_l$ of $\mathcal{G}_{[t_i,t_j]}$.

(ii) a disjunctive historical reachability query $u \overset{I_{Q\vee}}{\rightsquigarrow} v$ returns true, if there exists a path from u to v in at least one graph snapshot G_{t_m}, $t_k \leq t_m \leq t_l$, of $\mathcal{G}_{[t_i,t_j]}$.

Our goal is to derive methods for answering reachability queries efficiently. A straightforward solution would be to build a different index for each of the graph snapshots and then pose a reachability query at each one of them. However, this solution imposes large space overheads. In addition, it requires extra processing for combining the results of each query. Instead, we propose building indexes for intervals.

4. VERSION GRAPH

In this section, we present the version graph, a natural concrete representation of an evolving graph. First, let us define the notion of lifespan. For a node u (or, edge e), its lifespan denotes the set of time intervals during which u (resp. e) existed in an evolving graph. More formally, given an evolving graph $\mathcal{G}_I = \{G_{t_i}, G_{t_i+1}, \ldots, G_{t_j}\}$, the *lifespan*, $\mathcal{L}(u)$, (resp. $\mathcal{L}(e)$) of a node u (resp. edge e) is a set of intervals such that an interval $[t_i, t_j] \subseteq I$ belongs to $\mathcal{L}(u)$, (resp. $\mathcal{L}(e)$), if and only if, for all $t_i \leq t_m \leq t_j$, $u \in V_{t_m}$ (resp. $e \in E_{t_m}$).

We model lifespans as sets of time intervals to capture the general case of graph evolution, where nodes and edges may be deleted and then re-inserted at subsequent snapshots. Set of time intervals are also known as *temporal elements* [11]. If we do not allow deleted nodes or edges to be re-inserted, then lifespans are just intervals. Furthermore, if there are no deletions, all lifespans are intervals of the form $[t_i, t_{curr}]$, where t_i is the time instant the node or edge first appeared and t_{curr} is the time instant of the current snapshot. Therefore, in this case, lifespans can be represented simply by the time instant t_i. In the following, we use I to denote time intervals and $\mathcal{I}$ to denote sets of time intervals. To represent an evolving graph $\mathcal{G}_I$, we use a *version graph* VG_I. A version graph is a labeled directed graph that captures the evolution of the graph in a concise manner.

Definition 3 (version graph). *Given an evolving graph $\mathcal{G}_I = \{G_{t_i}, G_{t_i+1}, \ldots, G_{t_j}\}$, its version graph is an edge and node labeled, directed graph $VG_I = (V_I, E_I, \mathcal{L}_u, \mathcal{L}_e)$ where: $V_I = \bigcup_{t_m \in I} V_{t_m}$, $E_I = \bigcup_{t_m \in I} E_{t_m}$, $\mathcal{L}_u : V_I \to \mathcal{I}$ assigns to each node u in V_I its lifespan $\mathcal{L}_u(u)$ and $\mathcal{L}_e : E_I \to \mathcal{I}$ assigns to each edge e in E_I its lifespan $\mathcal{L}_e(e)$.*

An example is shown in Figure 1(b) which depicts the version graph for the evolving graph in Figure 1(a).

4.1 Lifespan Operations

Let us define a number of operations on lifespans, i.e., set of intervals. For two sets $\mathcal{I}$ and $\mathcal{I}'$ of time intervals, we say that $\mathcal{I}$ *covers* $\mathcal{I}'$, denoted $\mathcal{I} \sqsupseteq \mathcal{I}'$, if for each time instant t in an interval I' of $\mathcal{I}'$, there is an interval I in $\mathcal{I}$ such that t belongs to I. We also use $\mathcal{I} \sqsupseteq I$ for an interval I and $\mathcal{I} \sqsupseteq t$ for a time instant t. We say that two sets $\mathcal{I}$ and $\mathcal{I}'$ of time intervals are equivalent, $\mathcal{I} \approx \mathcal{I}'$, if $\mathcal{I} \sqsupseteq \mathcal{I}'$ and $\mathcal{I}' \sqsupseteq \mathcal{I}$.

We would like to maintain the smallest among equivalent sets of intervals. We call such sets *minimum* sets. Let us first define some simple properties for time intervals. Two time intervals $I = [t_i, t_j]$ and $I' = [t'_i, t'_j]$ are called *disjoint*, when $I \cap I' = \emptyset$ and *overlapping* otherwise. They are called *continuous* when $t'_i = t_j + 1$ and non-continuous otherwise. It is easy to see that the following proposition holds.

Proposition 1.

(i) A set of intervals is minimum, if and only if, it consists of disjoint and non-continuous intervals.

(ii) For each set of time intervals, there is a unique equivalent minimum interval set.

We next define two useful operations on interval sets, namely, *join* and *merge*. Given two sets of intervals, join returns the time instants common to both, while merge returns the time instants present in at least one of them.

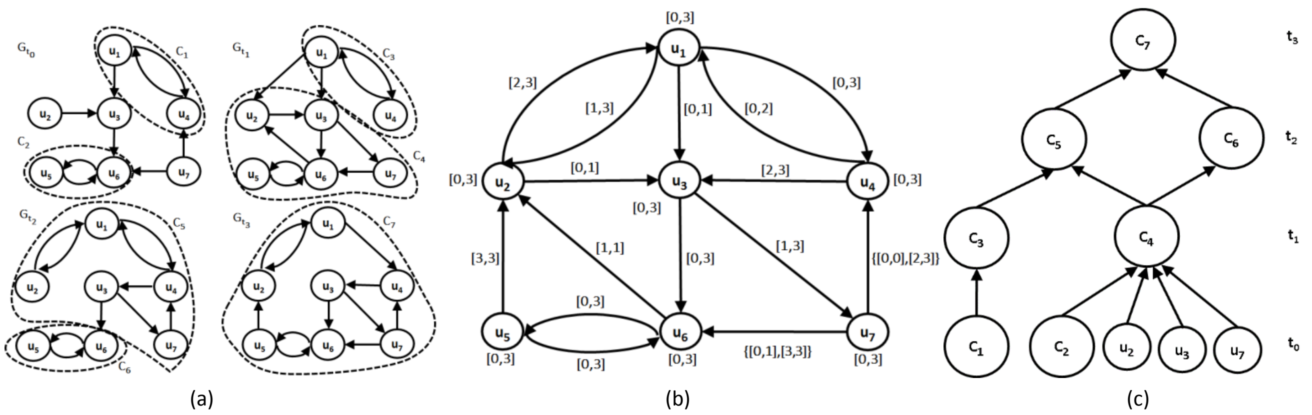


Figure 1: Example of (a) an evolving graph, (b) the corresponding version graph, (c) SCC evolution

Definition 4 (Join and Merge of Interval Sets). *Let $\mathcal{I} = \{I_1, \ldots\ I_k\}$ and $\mathcal{I}' = \{I'_1, \ldots\ I'_l\}$ be two sets of time intervals.*

(i) Join $\mathcal{I} \otimes \mathcal{I}'$ of $\mathcal{I}$ and $\mathcal{I}'$ is the minimum set equivalent to $\{I_1 \cap I'_1, \ldots\ I_1 \cap I'_l, \ldots, I_k \cap I'_1, \ldots\ I_k \cap I'_l\}$.

(ii) Merge $\mathcal{I} \oplus \mathcal{I}'$ of $\mathcal{I}$ and $\mathcal{I}'$ is the minimum set equivalent to $\mathcal{I} \cup \mathcal{I}'$.

Note that if $\mathcal{I}$ and $\mathcal{I}'$ are minimum, then the set $\{I_1 \cap I'_1, \ldots\ I_1 \cap I'_l \ldots, I_k \cap I'_1\ \}$ is a minimum set, whereas the set $\{I_1 \cup I'_1, \ldots\ I_1 \cup I'_l, \ldots, I_k \cup I'_1 \ldots\ I_k \cup I'_l\}$ may not be minimum.

The lifespan $\mathcal{L}(p)$ of a path p includes the time intervals during which all its edges coexist. Clearly, for a path $p = e_1 \ldots\ e_m$, it holds that $\mathcal{L}(p) = \mathcal{L}_e(e_1) \otimes \ldots \otimes \mathcal{L}_e(e_m)$, where $\mathcal{L}_e(e_i)$, $1 \leq i \leq m$, is the lifespan of e_i. For example, for path $p = ((u_4, u_3), (u_3, u_7), (u_7, u_6))$ in Figure 1(b), $\mathcal{L}(p) = \{[2,3]\} \otimes \{[1,3]\} \otimes \{[0,1], [3,3]\} = \{[3,3]\}$, while for path $p' = ((u_1, u_3), (u_3, u_7), (u_7, u_4))$, $\mathcal{L}(p') = \{[0,1]\} \otimes \{[1,3]\} \otimes \{[0,0], [2,3]\} = \emptyset$.

We can now define the lifespan, $\mathcal{L}(u,v)$, of the reachability between two nodes u and v. Let $P(u,v) = \{p_1, \ldots p_l\}$ be the set of all paths from u to v. $\mathcal{L}(u,v)$ depends on the lifespans of all possible paths in VG_I from u to v, in particular, $\mathcal{L}(u,v) = \mathcal{L}(p_1) \oplus \ldots \oplus \mathcal{L}(p_l)$. For example, for nodes u_4 and u_6 in Figure 1(b), $P(u_4, u_6) = \{p_1, p_2, p_3, p_4, p_5, p_6\}$ where $p_1 = u_4u_3u_6$, $p_2 = u_4u_3u_7u_6$, $p_3 = u_4u_1u_3u_6$, $p_4 = u_4u_1u_3u_7u_6$, $p_5 = u_4u_1u_2u_3u_6$, $p_6 = u_4u_1u_2u_3u_7u_6$ (note, that for notational brevity, paths were denoted by the participating nodes instead of edges). Then, $\mathcal{L}(u_4, u_6) = \{[2,3]\} \oplus \{[3,3]\} \oplus \{[0,1]\} \oplus \{[1,1]\} \oplus \{[1,1]\} \oplus \{[1,1]\} = \{[0,3]\}$.

Clearly, historical reachability queries can be represented in terms of lifespans. Specifically, given a version graph VG_I, a time interval $I_Q = [t_k, t_l] \subseteq [t_i, t_j]$ and two nodes v, u,

(i) a conjunctive historical reachability query $u \overset{I_{Q\wedge}}{\rightsquigarrow} v$ returns true, if and only if, $\{I_Q\} \otimes \mathcal{L}(u,v) \sqsupseteq I_Q$.

(ii) a disjunctive historical reachability query $u \overset{I_{Q\vee}}{\rightsquigarrow} v$ returns true, if and only if, $\{I_Q\} \otimes \mathcal{L}(u,v) \neq \emptyset$.

To represent lifespans, we use bit arrays. Assume without loss of generality, that the maximum time instant, that is, the number of graph snapshots, is T. Then, a lifespan, i.e., set of intervals, $\mathcal{I}$ is represented by a bit array B of size T, such that $B[i] = 1$ if $\mathcal{I} \sqsupseteq i$, and 0, otherwise. For example, take $\mathcal{I} = \{[2,4], [9,10], [13, curr]\}$ and $T = 16$. The bit array representation of $\mathcal{I}$ is 0011100001100111. This leads to an efficient implementation of both join $\otimes$ and merge $\oplus$. In particular, let $\mathcal{I}$ and $\mathcal{I}'$ be two set of intervals and B and B' be their bit arrays. Then, $\mathcal{I} \otimes \mathcal{I}'$ is computed as B logical-AND B' and $\mathcal{I} \oplus \mathcal{I}'$ as B logical-OR B'. An alternative representation would be to use ordered lists of intervals. Lifespan operations would then be performed using variations of merge sort resulting in $O(T)$ complexity. Lists impose in general large computational overheads in computing reachability.

4.2 Baseline Approaches

There are two baseline approaches to answering reachability queries on static graphs, namely pre-computation of the graph transitive closure and online traversal of the graph. In this section, we revisit these baseline approaches for historical reachability queries on a version graph.

4.2.1 Historical Transitive Closure

Instead of maintaining a different transitive closure for each graph snapshot of the evolving graph $\mathcal{G}_I$, we maintain a single transitive closure, CL_I for the version graph VG_I. The transitive closure includes for each pair of nodes u, v, their reachability lifespan, $\mathcal{L}(u,v)$. To construct the transitive closure, we use a variation of the Floyd-Warshall algorithm that takes into account lifespans, shown in Algorithm 1. If there is a path $p_{u,w}$ from node u to node w and a path $p_{w,v}$ from node w to node v then there exists a path $p_{u,v} = (p_{u,w}, p_{w,v})$ from u to v with $\mathcal{L}(p_{u,v}) = \mathcal{L}(p_{u,w}) \otimes \mathcal{L}(p_{w,v})$ and $\mathcal{L}(p_{u,v})$ is merged with the $\mathcal{L}(u,v)$ computed so far.

The time complexity for Algorithm 1 is $O(|V_I|^3 T)$ in the worst case and requires storage in the order of $|V_I|^2$. For answering a reachability query $u \overset{I_{Q\vee}}{\rightsquigarrow} v$ or $u \overset{I_{Q\wedge}}{\rightsquigarrow} v$, initially the entry $\mathcal{L}(u,v)$ in CL_I is located and then joined with the query interval I_Q, thus requiring constant time complexity.

4.2.2 Online Traversal of the Version Graph

A straightforward approach to process a reachability query for an interval I_Q would be to perform an online traversal on all graph snapshots G_t, $t \in I_Q$. When using the version graph representation, this corresponds to traversing

Algorithm 1 TransitiveClosure(VG_I)

Input: Version graph VG_I
Output: The transitive closure CL_I

```
1: for all $u$, $v \in V_I \times V_I$ do
2:   if $(u, v) \in E_I$ then
3:     $CL_I(u, v) = \mathcal{L}_e((u, v))$
4:   else
5:     $CL_I(u, v) = \emptyset$
6:   end if
7: end for
8: for $w = 1$ to $|V_I|$ do
9:   for all $u$, $v \in V_I \times V_I$ do
10:    $CL_I(u, v) = CL_I(u, v) \oplus (CL_I(u, w) \otimes CL_I(w, v))$
11:  end for
12: end for
```

Algorithm 2 Disjunctive-BFS(VG_I, u, v, $\{I_Q\}$)

Input: Version graph VG_I, nodes u, v, interval $I_Q \subseteq I$
Output: True if v is reachable from u in any time instant in I_Q and false otherwise

```
1: create a queue $N$, create a queue $INT$
2: enqueue $u$ onto $N$, enqueue $I_Q$ onto $INT$
3: while $N \neq \emptyset$ do
4:   $n \leftarrow N.dequeue()$
5:   $i \leftarrow INT.dequeue()$
6:   for all $w$ s.t. $(n, w)$ in $VG_I$ and $\{I_Q\} \otimes \mathcal{L}_e((n, w)) \neq \emptyset$ do
7:     if w == $v$ then
8:       Return(true)
9:     end if
10:    $\mathcal{I}' = \{I_Q\} \otimes \mathcal{L}_e(u, w)$
11:    if $\mathcal{IN}(w) \not\sqsupseteq \mathcal{I}'$ then
12:      $\mathcal{IN}(w) = \mathcal{IN}(w) \oplus \mathcal{I}'$
13:      enqueue $w$ onto $N$
14:      enqueue $\mathcal{I}'$ onto $INT$
15:    end if
16:  end for
17: end while
18: Return(false)
```

only edges e such that $\mathcal{L}_e(e) \sqsupseteq t$, once for each $t \in I_Q$. We call this approach, *instant based traversal*.

To avoid multiple traversals, i.e., one for each snapshot in I_Q, we consider an *interval based traversal* of the version graph. The BFS-based interval traversal for disjunctive historical queries is shown in Algorithm 2 and for conjunctive historical queries in Algorithm 3.

In particular, for conjunctive queries, since a node v may be reachable from u through different paths at different graph snapshots, we maintain an interval set $\mathcal{R}$ with the part of $\mathcal{L}(u, v) \otimes I_Q$ covered so far (line 9, Algorithm 3). The traversal ends when $\mathcal{R}$ covers the whole query time interval I_Q (line 10, Algorithm 3).

To speed-up traversal, we perform a number of pruning tests. The traversal stops when we reach a node whose lifespan is outside the query interval. In addition, the traversal stops at a neighbor w of a node n when $\{I_Q\} \otimes \mathcal{L}_e(n, w) \neq \emptyset$ since a node v cannot be reachable through an edge which is not alive in at least one t inside the query interval (line 6, Algorithms 2 and 3).

Still an edge may be traversed multiple times, if it participates in multiple paths from source to target. To reduce the number of such traversals, we provide additional pruning by recording for each node w, an interval set $\mathcal{IN}(w)$ with the parts of the query interval for which it has already been traversed. If the query reaches w again looking for interval I' $\subseteq I_Q$ and $\mathcal{IN}(w) \sqsupseteq I'$, the traversal is pruned (line 11 of Algorithm 2, line 15 of Algorithm 3).

For example, consider the version graph in Figure 1(b) and query $u_1 \overset{[0,3]\wedge}{\rightsquigarrow} u_5$. Paths $p_1 = u_1u_3u_6u_5$, $p_2 = u_1u_3u_7u_6u_5$, $p_3 = u_1u_2u_3u_6u_5$, $p_4 = u_1u_2u_3u_7u_6u_5$, $p_5 = u_1u_4u_3u_6u_5$ and $p_6 = u_1u_4u_3u_7u_6u_5$ with $\mathcal{L}(p_1) = \{[0, 1]\}$, $\mathcal{L}(p_2) = \{[1, 1]\}$, $\mathcal{L}(p_3) = \{[1, 1]\}$, $\mathcal{L}(p_4) = \{[1, 1]\}$, $\mathcal{L}(p_5) = \{[2, 3]\}$ and $\mathcal{L}(p_6)$ = $\{[3, 3]\}$ need to be traversed to conclude correctly that the result of the query is true. Hence, some edges, e.g., (u_3, u_6), (u_6, u_5) need to be traversed multiple times for different time intervals $I'_i \subseteq I_Q$. However, when the query reaches u_3 again through path p_3, it is pruned and it does not traverse the edge (u_3, u_6) since $\mathcal{IN}(u_3)$ is equal to {[0,1]} which covers the current query interval I' = {[1,1]}.

Since in the worst case for both instant and interval based traversal each edge may be traversed $|I_Q|$ times, the complexity for both traversals is $O((|V_I| + |E_I|)|I_Q|)$. However, in practice interval based traversal outperforms the instant based one since each edge traversal covers large parts of the query interval instead of a single time instant. Furthermore, pruning guarantees that an edge will not be traversed twice for the same interval.

5. THE TIMEREACH INDEX

Our approach exploits the fact that many real-world social graphs are characterized by large strongly connected components (SCC) [20, 15]. Thus, instead of maintaining reachability information for pairs of nodes, we maintain information about the SCCs that each node belongs to. If two nodes belong to the same component, then they are reachable. However, as the graph evolves over time, its strongly connected components change as well. An example is shown in Figure 1(c) that depicts the SCCs of the graph in Figure 1(b) as they evolve over time.

Given an evolving graph $\mathcal{G}_I = \{G_{t_i}, G_{t_i+1}, \ldots, G_{t_j}\}$, we invoke at each graph snapshot $G_{t_k} \in \mathcal{G}_I$ an algorithm, e.g., Tarjan's algorithm [24], to identify the corresponding set of SCCs. A unique id is assigned to each SCC at each snapshot.

For each node u, we maintain a list $P(u)$ that contains (C, t) pairs specifying the strongly connected component C to which node u belongs at time instant t. $P(u)$ is called *posting list* and each pair in the list a *posting*. The storage complexity is $\Omega(|V_I||I|)$, since each node participates in at most one SCC at each time instant. If we use Tarjan's algorithm [24], the time complexity for constructing the lists is $O((|V_I| + |E_I|)|I|)$, since each run of the Tarjan's algorithm has an $O(|V_I| + |E_I|)$ complexity.

For presentation clarity, we assume that single nodes form singleton SCCs whose ids are the ids of the corresponding nodes. However, for space efficiency, we do not maintain postings in this case.

We perform an additional optimization. Many nodes have strong connections, i.e. they remain in the same components even in the face of component splits and joins. We exploit this fact to reduce the storage space required for the postings by observing that the posting lists of these nodes consist of

Algorithm 3 Conjunctive-BFS(VG_I, u, v, $\{I_Q\}$)

Input: Version graph VG_I, nodes u, v, interval $I_Q \subseteq I$
Output: True if v is reachable from u in all time instants in I_Q and false otherwise

```
1: create a queue $N$, create a queue $INT$
2: enqueue $u$ onto $N$, enqueue $I_Q$ onto $INT$
3: while $N \neq \emptyset$ do
4:   $n \leftarrow N.dequeue()$
5:   $i \leftarrow INT.dequeue()$
6:   for all $w$ s.t. $(n, w)$ in $VG_I$ and $\{I_Q\} \otimes \mathcal{L}_e((n,w)) \neq \emptyset$ do
7:     $\mathcal{I}' = \{I_Q\} \otimes \mathcal{L}_e(n,w)$
8:     if $w == v$ then
9:       $R = R \oplus \mathcal{I}'$
10:      if $\mathcal{R} \sqsupseteq I_Q$ then
11:        Return(true)
12:      end if
13:      continue
14:    end if
15:    if $\mathcal{IN}(w) \not\sqsupseteq \mathcal{I}'$ then
16:      $\mathcal{IN}(w) = \mathcal{IN}(w) \oplus \mathcal{I}'$
17:      enqueue $w$ onto $N$
18:      enqueue $\mathcal{I}'$ onto $INT$
19:    end if
20:  end for
21: end while
22: Return(false)
```

the same elements. We avoid redundancy by storing such lists only once and replacing the posting lists of the relevant nodes with pointers to the common list. We call this approach *posting sharing.*

An example is shown in Figure 2(a), where, for instance, the first posting list indicates that nodes with ids 1 up to 50 belong to the strongly connected component with id C_1 at time t_0, C_6 at t_1 and C_9 at t_2.

In addition, for each graph snapshot G_{t_k}, we construct a SCC graph snapshot $G_{S_{t_k}} = (V_{S_{t_k}}, E_{S_{t_k}})$ such that there is a node U in $V_{S_{t_k}}$ for each SCC in G_{t_k}, and there is an edge (U, V) in $E_{S_{t_k}}$, if and only if, there is an edge (u,v) in G_{t_k} from a node u that belongs to the SCC that corresponds to U to a node v that belongs to the SCC that corresponds to V. For a time interval $I = [t_i, t_j]$, this results in an evolving SCC graph $\mathcal{G}_{S_I} = \{G_{S_{t_i}}, G_{S_{t_i+1}}, \ldots, G_{S_{t_j}}\}$. We construct the SCC graphs incrementally, as the SCCs are created. The size of each SCC graph depends on the size of the original snapshot graph and in the worst case is equal to it.

We call this approach simple *TimeReach* (TR). To answer a reachability query $u \overset{I_{Q\wedge}}{\rightsquigarrow} v$, (or, $u \overset{I_{Q\vee}}{\rightsquigarrow} v$), we check for each $t \in I_Q$ whether u and v belong to the same component. If this is not the case, we traverse the corresponding G_{S_t}.

Next, we present a more space efficient method of exploiting strongly connected components for historical queries.

5.1 Condensed TimeReach

While in the TR approach, we maintain information per time instant, we would like to aggregate such information to express SCC participations during time intervals. In this case, a posting (C, I'), $I' \subseteq I$, belongs to $P(u)$, if u participates in the SCC with id C at all time instants in I'. Our goal is to minimize the total number of such postings.

Problem 1 (Optimal SCC-id assignment). *Given a time interval I and a set of SCCs for each $t \in I$, find an assignment of ids to SCCs that results in the minimum number of postings.*

A new posting is created, each time a node participates in a different SCC. Thus, SCC ids should be re-assigned so that the number of such new postings is minimized. We use a weighted graph to formalize the optimal assignment of ids to SCCs.

In particular, we model SCC evolution over a time interval I using a weighted graph $G_C(V_C, E_C, \mathcal{W})$ where each node $U \in V_C$ corresponds to a SCC that existed at some time instant $t \in I$, and an edge $e = (U, V) \in E_C$, if and only if, SCC U existed at time t_k, SCC V existed at time $t_k + 1$ and there is at least one node that belongs to both U and V. $\mathcal{W}$ assigns to each edge $e = (U, V)$ a weight $\mathcal{W}(e)$ that corresponds to the number of nodes that belong to both U and V.

An example of a weighted graph is shown in Figure 2(b) that depicts the evolution of the graph whose posting lists are shown in Figure 2(a). For instance, component C_7 created at time instant t_1 consists of 100 nodes from component C_4 and 150 nodes from C_5.

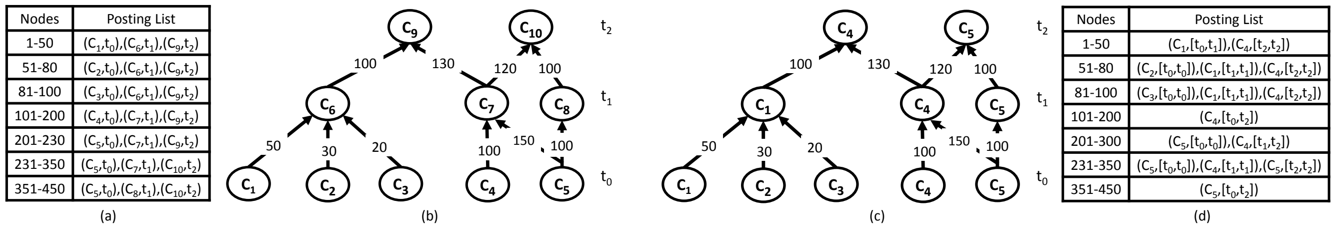
Let $G_{C_{[t_k,t_k+1]}}(V_{C_{[t_k,t_k+1]}}, E_{C_{[t_k,t_k+1]}}, \mathcal{W})$ be the subgraph of $G_C(V_C, E_C, w)$, that consists of the nodes $U \in V_{C_{[t_k,t_k+1]}}$ that correspond to the SCCs that exist at time interval $[t_k, t_k+1]$. $G_{C_{[t_k,t_k+1]}}$ represents one step in the SCC evolution. Note that, from the definition of G_C, $G_{C_{[t_k,t_k+1]}}$ is a bipartite graph.

We make the following observation. At time instant $t_k + 1$, a new posting is created exactly for those nodes that participated in a different SCC at $t_k + 1$ than at t_k. The number of these new postings is equal to the sum of weights from node U to V in $G_{C_{[t_k,t_k+1]}}$ where U has a different id than V. Thus, to minimize the number of new postings, we have to maximize the weight of the edges between pairs of nodes that have the same id. This corresponds to finding a maximum bipartite matching of $G_{C_{[t_k,t_k+1]}}$.

Theorem 1. *The optimal SCC-id assignment problem can be reduced to the problem of finding the maximum weight bipartite matching (MWM) M_k of each $G_{C_{[t_k,t_k+1]}}$.*

Proof. As shown above, solving the MWM for each bipartite graph $G_{C_{[t_k,t_k+1]}}$ minimizes the number of new postings created at $t_k + 1$. We shall show that this step-wise assignment is optimal overall in G_C. For the purposes of contradiction, assume that the optimal assignment is a set N of edges, $N \subset E_C$ and that N is different from the set of edges attained through the maximum bipartite matchings, that is, $\sum_{e\in N} w(e) > \sum_k \sum_{e\in M_k} w(e)$. Hence, for some m, for $N_m = N \cap E_{C_{[t_m,t_m+1]}}$ it holds that $\sum_{e\in N_m} w(e) > \sum_{e\in M_m} w(e)$, which means that M_m is not a MWM, which is a contradiction. □

Figure 2(c) shows the weighted graph after the assignment of new ids through bipartite matching, while Figure 2(d) shows the new posting lists.



Nodes	Posting List
1-50	$(C_1,t_0),(C_6,t_1),(C_9,t_2)$
51-80	$(C_2,t_0),(C_6,t_1),(C_9,t_2)$
81-100	$(C_3,t_0),(C_6,t_1),(C_9,t_2)$
101-200	$(C_4,t_0),(C_7,t_1),(C_9,t_2)$
201-230	$(C_5,t_0),(C_7,t_1),(C_9,t_2)$
231-350	$(C_5,t_0),(C_7,t_1),(C_{10},t_2)$
351-450	$(C_5,t_0),(C_8,t_1),(C_{10},t_2)$

(a)

Nodes	Posting List
1-50	$(C_1,[t_0,t_1]),(C_4,[t_2,t_2])$
51-80	$(C_2,[t_0,t_0]),(C_1,[t_1,t_1]),(C_4,[t_2,t_2])$
81-100	$(C_3,[t_0,t_0]),(C_1,[t_1,t_1]),(C_4,[t_2,t_2])$
101-200	$(C_4,[t_0,t_2])$
201-300	$(C_5,[t_0,t_0]),(C_4,[t_1,t_2])$
231-350	$(C_5,[t_0,t_0]),(C_4,[t_1,t_1]),(C_5,[t_2,t_2])$
351-450	$(C_5,[t_0,t_2])$

(d)

Figure 2: (a) Shared posting lists, (b) weighted graph modeling the evolution of SCCs, (c) weighted graph after the bipartite matching, and (d) the compressed shared posting lists

The maximum weight bipartite matching problem is well-studied (e.g., see [8] for a survey). The most widely used algorithm for solving this problem on a graph $G(V, E)$ is the Hungarian algorithm whose running time ranges from $O(|V|^3)$ to $O(|E||V| + |V|^2 loglog|V|)$ depending on the implementation. Another category of algorithms depends on the edge weights and the fastest one runs in $O(|E|\sqrt{|V|}logW)$ time, where W is the maximum edge weight. In addition, a number of fast approximation algorithms have been proposed. The simplest such algorithm is the greedy algorithm that sorts the edges by weight and repeatedly picks the edge with the largest weight. This algorithm can be implemented with $O(|E|)$ time complexity and produces a 1/2 worst case approximation.

The incremental algorithm for constructing the SCC postings is presented in Algorithm 4. It takes as input the current snapshot and the postings computed up to the previous snapshot, and constructs the current postings. It starts by computing the SCCs using Tarjan's algorithm with complexity $O(|V_t| + |E_t|)$ (line 2). Then, it constructs the graph $G_{C_{[t,t+1]}}$ with complexity $O(|E_{C_{[t-1,t]}}|)$ (line 5). Next, the MWM is computed and new ids are assigned to the new SCCs (lines 6 - 9). The complexity of this step depends on which algorithm is used for computing the MWM. We use the greedy algorithm with complexity $O(|E_{S_{[t-1,t]}}|)$. Finally, the SCC postings are created/updated for each node of the current snapshot, creating a new entry only for nodes that participate in a different SCC (with a different id) than the one in time instant $t - 1$ (lines 11 - 22). The complexity of these steps is $O(|V_t|)$ since each operation in the loop has constant time complexity. Thus, in total the running time of the algorithm is $O(|V_t| + |E_t|)$.

As in the simple TR approach, we also construct the evolving SCC graph, which in this case has a much smaller number of nodes due to the reduction of the number of strongly connected components achieved by the bipartite matching.

Finally, we construct the version graph $VG_{S_I} = (V_{S_I}, E_{S_I}, \mathcal{L}_u, \mathcal{L}_e)$ of the evolving SCC graph that we call *condensed version graph.* We construct the condensed version graph incrementally as follows. For each snapshot $G_{t_i} \in \mathcal{G}_I$, for each edge $(u, v) \in E_{t_i}$ we look up the postings $P(u)$, $P(v)$ for entries (U, I'), (V, I'') s.t. $t_i \in I'$ and $t_i \in I''$. If $U \neq V$ and edge $(U, V) \notin E_{S_I}$, the edge is added with lifespan $\{[t_i, t_i]\}$, otherwise the lifespan of the edge is extended to include t_i. We call the above approach *condensed TimeReach* (TRC).

5.2 Query Processing

Query processing of a (disjunctive or conjunctive) reachability query $u \overset{I_Q}{\leadsto} v$ is performed in two steps. In the first step, the appropriate postings of nodes u and v are retrieved. If the two nodes belong to the same strongly connected component during the whole query interval for conjunctive queries or once for disjunctive queries, the answer is true. Otherwise, let $\mathcal{I}'_Q$ be the set of intervals during which nodes u and v belong to different components. The query is re-written as a set of reachability sub-queries of the form $U_k \overset{I_{Q_i}}{\leadsto} V_m$, where u belongs to SCC U_k and v belongs to SCC V_m for some common time interval I_{Q_i}, $\mathcal{I}'_Q \sqsupseteq I_{Q_i}$, the set $\mathcal{I}_Q = \bigcup_i I_{Q_i}$ consists of disjoint intervals, and $\mathcal{I}_Q \approx \mathcal{I}'_Q$. The results of the sub-queries are combined to produce the answer for the query through an AND (OR) for conjunctive (disjunctive) queries.

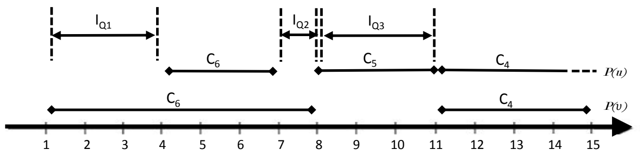


Figure 3: Example of splitting query $u \overset{[1,15]\wedge}{\leadsto} v$

For example, consider the query $u \overset{[1,15]\wedge}{\leadsto} v$ in Figure 3, where the posting lists for u and v are respectively, $P(u) = (C_6\ [4, 7],\ C_5\ [8, 11],\ C_4\ [11, curr]$ and $P(v) = (C_6\ [1, 8],\ C_4\ [11, 15])$. The query is split in three sub-queries: $u \overset{I_{Q1}\wedge}{\leadsto} C_6$, $u \overset{I_{Q2}\wedge}{\leadsto} C_6$, $v \overset{I_{Q3}\wedge}{\leadsto} C_5$.

In the worst case, the two nodes belong to a different SCCs at each time instant in I_Q, thus we need to traverse the condensed version graph for each t with a cost of $O(|I_Q|(|V_{S_I}| + |E_{S_I}|))$ Two factors that influence performance are the number of postings for each node and the size of the condensed version graph. The smaller the number of postings, the fewer sub-queries are required in the second step. The smaller the condensed version graph, the faster the traversals. Hence, the optimal assignment of SCC ids is crucial to query processing performance, since it keeps the posting lists short and the size of the condensed version graph small.

5.3 Interval 2Hop

Reachability on version graphs can be made more efficient by maintaining additional information. In this paper, we use an approach based on pruned landmark 2hop labeling [2, 29]. The idea is that for each node u of a given graph, we maintain two labels $L_{in}(u)$ and $L_{out}(u)$ which include nodes that can reach u and can be reached by u respectively. The labels are computed such that a node u reaches v, if an only if, $L_{in}(v) \cap L_{out}(u) \neq \emptyset$. Instead of traversing the graph, a reachability query can now be answered by using the labels.

For historical reachability queries, we also keep along with each node w in $L_{in}(v)$ the reachability lifespan $\mathcal{L}(w, v)$ and along with each node w in $L_{out}(u)$ the reachability lifespan

Algorithm 4 ConstructSccPostings(G_t, P_{t-1}, $G_{S_{[t-2,t-1]}}$)

Input: Snapshot G_t, SCC postings P_{t-1}
Output: SCC postings P_t

```
 1: $S_{SCC_t} = \emptyset$, $M = \emptyset$
 2: Run Tarjan's algorithm on $G_t$
 3: $S_{SCC_t}$ is the set of the detected SCCs where each
    $SCC_i \in S_{SCC_t}$ is assigned a unique id $C_i$
 4: if $t > 0$ then
 5:    Construct $G_{S_{[t-1,t]}}$ from $S_{SCC_t}$ and $G_{S_{[t-2,t-1]}}$
 6:    Compute maximum weight matching $M$
 7:    for all edges $e = (U, V) \in M$ do
 8:       $C_v = C_u$
 9:    end for
10: end if
11: for all nodes $u \in V_t$ do
12:    find $SCC_i \in S_{SCC_t}$ s.t. $u \in SCC_i$
13:    if $P_{t-1}(u) \neq \emptyset$ then
14:       if $P_{t-1}(u)[end].C \neq C_i$ then
15:          $P_{t-1}(u)[end].I = [t_s, t-1]$
16:          $P_{t-1}(u).add(C_i, [t, curr])$
17:       end if
18:    else
19:       $P_{t-1}(u).add(C_i, [t, curr])$
20:    end if
21: end for
22: $P_t = P_{t-1}$
```

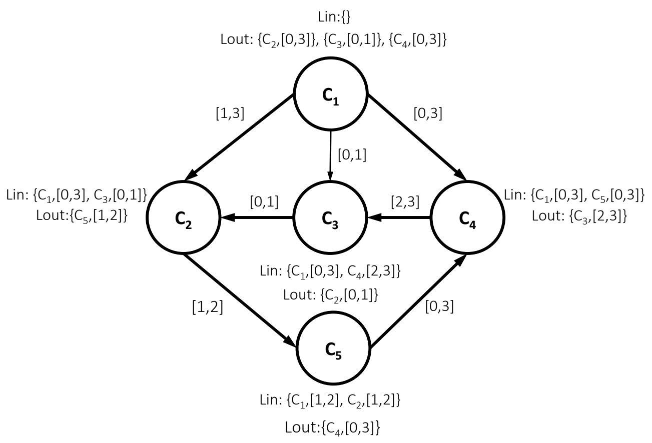
$\mathcal{L}(u, w)$. In the presence of 2hop labels, to answer a query $u \overset{I_Q}{\leadsto}_{\wedge} v$ ($u \overset{I_Q}{\leadsto}_{\vee} v$), we compute the set $L_{in}(v) \cap L_{out}(u)$ and then for each w in $L_{in}(v) \cap L_{out}$, we join the lifespan of w in $L_{in}(v)$ with the lifespan of w in $L_{out}(u)$. To answer the query the joined lifespans $\mathcal{L}$ (w) of nodes w in $L_{in}(v) \cap L_{out}$ are joined with the query interval $\mathcal{L}$ to see whether they cover I_Q (or, have at least a time instant in common).

We compute the labels for the nodes of the condensed version graph, incrementally. For an interval $I = [t_i, t_j]$, we compute the labels for the SCC graph snapshots at each time t in I, starting from t_i. For each time t_k, $t_k > t_i$, we merge the labels computed for a node C at time t_k, with the labels computed for C at the previous time $t_k - 1$. For the construction of L_{in} and L_{out} for each SCC graph snapshot at time instant t_k, we process the nodes of the graph by using the $INOUT$ strategy that starts a BFS traversal from the nodes with the largest $(indegree(u){+}1) \times (outdegree(u){+}1)$ [29]. An example of the final 2hop labels of each SCC node in a version graph is given in Figure 4.

6. EXPERIMENTAL EVALUATION

To evaluate our approach, we used three real datasets: Facebook (FB) [27], Flickr (FL) [19] and YouTube (YT) [18]. The characteristics of each dataset are shown in Table 3. For example, FB consists of 871 daily snapshots of the New Orleans Facebook friendship graph, which correspond to 125 weekly or 29 monthly snapshots. We report the number of nodes, edges, and SCCs (singleton SCCs are not included) and the size of the largest SCC at the first and last snapshot.

All three datasets are treated as directed. Also, all datasets are insert-only, i.e. they do not contain information about node/edge deletions. Therefore, we synthetically generate random edge deletes. The input parameters and their de-



Lin:{}
Lout: {C1,[0,3]}, {C3,[0,1]}, {C4,[0,3]}
C1
[1,3]
[0,3]
[0,1]
Lin: {C1,[0,3], C3,[0,1]}
Lout:{C5,[1,2]}
C2
[0,1]
C3
[2,3]
C4
Lin: {C1,[0,3], C5,[0,3]}
Lout: {C3,[2,3]}
Lin: {C1,[0,3], C4,[2,3]}
Lout: {C2,[0,1]}
[1,2]
[0,3]
C5
Lin: {C1,[1,2], C2,[1,2]}
Lout:{C4,[0,3]}

Figure 4: An example of interval 2hop labels

fault values are shown in Table 1.

We evaluate the size and the construction time of the Version Graph (VG), the Transitive Closure (TC), the simple TimeReach (TR), the condensed TimeReach (TRC) and the condensed TimeReach with 2hop labels (TRCH). We also evaluate the online processing of historical reachability queries using an instant-based (INS) or interval-based (INT) traversal of the version graph and using the various TimeReach indexes. Table 2 summarizes the various approaches.

We ran our experiments on a system with a quad-core Intel Core i7-3820 3.6 Ghz processor and 64 GB memory. We only used one core for all experiments.

Table 1: Input parameters

		# of nodes	**Snapshot granularity**	**Query interval (in days)**	**% of deletes**
FB	Default	61,096	day	7	10
	Range	117 - 61,096	day, week, month	7 - 35	0 - 30
YT	Default	1,138,499	day	7	10
	Range	1,004,777 - 1,138,499	day, week, month	7 - 35	0 - 30
FL	Default	2,302,925	day	7	10
	Range	1,487,058 - 2,302,925	day, week, month	7 - 35	0 - 30

6.1 Index Size

In the first set of experiments, we evaluate the various approaches in terms of their storage requirements. The size of the TR and TRC include the storage required for maintaining the posting lists and the SCC graphs, while the size of the TRCH includes in addition the storage required for the 2hop labels.

Table 2: Overview of difference approaches

VG	Version Graph
TC	Transitive Closure
TR	(Simple) TimeReach
TRC	Condensed TimeReach
TRCH	Condensed TimeReach with 2hop labels
INS	Instant-based traversal of the version graph
INT	Interval-based traversal of the version graph

Table 3: Dataset properties

	Snapshot Granularity	# nodes		# edges		# SCC		Max SCC (# nodes)	
		first	last	first	last	first	last	first	last
FB	(daily) 871	117	61,096	128	1,139,081	10	374	3	51,286
	(weekly) 125	1,429	61,096	2,365	1,139,081	138	374	18	51,286
	(monthly) 29	4,239	61,096	12,224	1,139,081	279	374	96	51,286
YT	(daily) 37	1,004,777	1,138,499	4,379,283	4,452,646	9,807	11,360	457,932	509,332
	(weekly) 6	1,025,536	1,138,499	4,379,283	4,452,646	9,807	11,360	465,668	509,332
	(monthly) 2	1,116,602	1,138,499	4,446,042	4,452,646	10,664	11,360	485,273	509,332
FL	(daily) 134	1,487,058	2,302,925	17,022,083	33,140,018	42,163	58,636	1,004,426	1,605,184
	(weekly) 20	1,507,700	2,302,925	17,393,321	33,140,018	42,163	58,636	1,010,498	1,605,184
	(monthly) 5	1,585,173	2,302,925	18,987,847	33,140,018	42,459	58,636	1,081,499	1,605,184

Graph Size (scalability). Figure 6 reports the size for varying number of nodes. As shown, TRC is much smaller than TR in all cases. For FB and FL, the largest SCC covers 83% and 70% of the graph respectively, while for YT, it covers just 45% (see Table 3). Thus, the TRC size for the FB dataset is 89% smaller, while for the YT and FL datasets, we achieve 40% and 57% of compression respectively. The larger the SCCs, the higher the compression achieved.

Since the size of the transitive closure (TC) grows rapidly, we compute TC for a smaller subset of the FB dataset varying the number of nodes from 1,000 to 6,000. As shown in Table 4, even for this small graph, the size of TC reaches 106 MB.

Percentage of Deletes. For each dataset, we vary the percentage of edge deletes from 0% to 30% of edge insertions. Table 5 presents the results for the FB dataset. We observe that the size of TR and TRC decreases; this can be explained by the fact that deletions affect the isolated nodes that become disconnected from the components and thus there are less edges between components and isolated nodes. The size of VG remains constant, since the size of the lifespan labels remains the same. Finally, the size of TRCH increases, because in case of deletes, additional nodes need to be included in the 2hop labels for ensuring the reachability test.

Table 4: Comparison with transitive closure

# nodes	Size (MB)			Constr. Time (sec)		
	TR	TRC	TC	TR	TRC	TC
1,000	0.013	0.012	2.91	0.01	4.76	167.49
2,000	0.026	0.009	11.56	0.23	5.02	1,457
3,000	0.039	0.012	26.27	0.35	5.89	5,788
4,000	0.052	0.018	47.12	0.41	6.33	16,580
5,000	0.063	0.026	73.97	0.59	6.79	39,112
6,000	0.074	0.032	106.82	0.72	7.13	81,123

Snapshot Granularity. Table 6 reports the storage required for maintaining daily, weekly and monthly snapshots of the three datasets. All sizes increase with the number of snapshots. For example, for FL, the increase of the number of snapshots by a factor of 30 (from 5 monthly to 134 daily) causes an increase of the size of TR by a factor of 3.44. The size of TR and TRC decreases with the snapshot granularity (number of snapshots) since less snapshots mean less postings and smaller SCC graphs. The size of VG

Table 5: Size per % of deletes (Facebook)

% of deletes	Size (MB)			
	VG	TR	TRC	TRCH
0	11	0.5	0.21	1,493
10	11	0.58	0.22	1,528
20	11	0.45	0.19	1,612
30	11	0.47	0.18	1,664

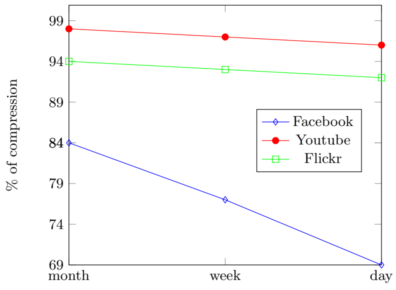


Figure 5: Compression ratio achieved by posting sharing

does not decrease significantly, because it requires memory to keep lifespan labels for all nodes and edges of the graph.

Posting Sharing. Finally, let us take a closer look at the posting sharing optimization by evaluating the reduction in the size of postings for various granularities as depicted in Figure 5. In general, we achieve compression ratios for the posting around 70% for FB, around 90% for FL and over 95% for YT. The compression ratio decreases with snapshot granularity due to the increase of the posting combinations. This is more evident for the FB dataset where the number of snapshots is higher.

6.2 Construction Time

In this set of experiments, we evaluate the time to construct the various indexes.

As seen in Figure 7, TRC is slower than TR, because of the additional time required for performing the bipartite matching. TRCH is even slower, since it also needs to construct the 2hop labels. We use the greedy algorithm for the bipartite matching and the INOUT strategy for computing the interval-2hop labels.

Constructing the TC for the whole graphs is prohibitive, since even for only 6,000 nodes, it takes over 22 hours, while

Table 6: Size (MB) per snapshot granularity

	Facebook			*YouTube*			*Flickr*		
	Days	Weeks	Months	Days	Weeks	Months	Days	Weeks	Months
VG	11	6	5	7.87	7.34	6.94	45.52	39.85	38.15
TR	0.58	0.47	0.42	44.28	21.28	14.98	141	73	41
TRC	0.22	0.08	0.07	3.21	1.92	1.46	2.89	2.27	1.88
TRCH	1,528	1,041	845	5,865	4,936	4,062	7,951	6,684	5,719

the TR construction takes just 0.72 seconds (Table 4).

Comparison of Different Bipartite Matching Algorithms. We also constructed the TRC using the Hungarian algorithm. For all datasets, the size of the resulting TRC is almost equal to the size of the TRC resulting from using the greedy algorithm (the difference is in the order of KB), thus confirming our expectation that greedy achieves a very close approximation of the optimal solution for social graphs. The Hungarian algorithm is much slower than greedy requiring an additional 1.5 hour for large datasets such as FL.

Comparison with 2hop for insert only. We adopted the pruned labeling algorithm proposed in [2] for distance queries to create an indexing scheme for historical reachability queries. Pruned labeling incrementally updates the index for each newly inserted edge, whereas in our approach we compute 2hop labels per snapshot. The pruned labeling algorithm does not support deletions, thus, we compare the two algorithms on the Facebook dataset without deletions. The pruned algorithm was found to be 5.4 times faster but it produced labels that were 12 times larger that the ones computed with our approach.

6.3 Query Processing

Let us now focus on query processing. In each experiment, we ran 500 historical reachability queries where the source and target nodes are chosen uniformly at random with the restriction that both nodes are present in the graph at the beginning and the end of the query interval. Queries involving nodes not present either at the beginning or the end of the query interval can be pruned fast by checking the lifespans of the nodes.

Online Traversal of the Version Graph. Let us first compare between an instant-based (INS) and an interval-based (INT) online traversal of the version graph for different time intervals (Figures 8 and 9). A general remark that holds independently of the method used to evaluate queries is that false conjunctive queries are faster than true conjunctive queries, since processing stops as soon as a time instant is found at which the two nodes are not reachable. Analogously, true disjunctive queries are faster than false disjunctive queries, since processing stops as soon as a time instant is found at which the two nodes are reachable.

Interval-based traversal is faster that instant-based traversal for almost all datasets and query types, since it can find the answer faster by searching for longer intervals. The only exception is FB and false conjunctive queries, where INS is slightly better. This happens because with INS, the search stops as soon as the first false answer is produced in any traversal. Hence, if this answer is found in the first few time instants of the query interval negative answers can be produced quickly for the smaller graph (i.e, the FB graph).

Online Traversal versus TimeReach. Let us now compare interval-based online traversal with query processing using the TR, TRC and TRCH approaches. The results for conjunctive queries are shown in Figure 10 and for disjunctive queries in Figure 11.

We see that all approaches are not significantly affected by the increase of the query interval due to fast posting lookups and short distances in the SCC graph for the TR and TRC, and the efficient implementation of edge lifespans for the version graph. We see that the TRC approach does not only produce a smaller structure than TR but it also attains faster query response for almost all datasets. TR is slower because for answering a query it needs to traverse the SCC graph per time instant when the query nodes do not belong to the same component. TRCH attains the fastest time when compared with all other approaches. The performance of TRCH is expected, since only two simple steps are needed: first to obtain the intersection $L_{in}(v) \cap L_{out}(u)$, and after that to check the lifespans $\mathcal{L}$ of the nodes in the intersection.

7. CONCLUSIONS

Most real-life graphs evolve over time. In this paper, we address the problem of efficiently answering historical reachability queries over such graphs. Such queries ask whether a node u was reachable from another node v during a time interval in the past. We have proposed an approach termed *TimeReach* that exploits the fact that most graphs consist of strongly connected components (SCCs). TimeReach maintains information about SCC membership for each node, and a graph which represents the links between the strongly connected components. We also maintain a condensed version graph which corresponds to the version graph of the SCCs evolution. Our extensive experiments with three real social network datasets show that TimeReach is storage-efficient and can be constructed incrementally with a small overhead. Historical queries are processed efficiently even when involving large time intervals.

There are many possible directions for future work. One such direction is exploiting TimeReach towards answering other types of historical queries, such as shortest path ones. Another direction concerns the distribution of TimeReach. Distribution may either be based on time or exploit the SCC evolution by placing together nodes that belong to the same SCCs.

8. ACKNOWLEDGMENTS

This research has been co-financed by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program ”Education and Lifelong Learning” of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thales. Investing in knowledge society through the European Social Fund.

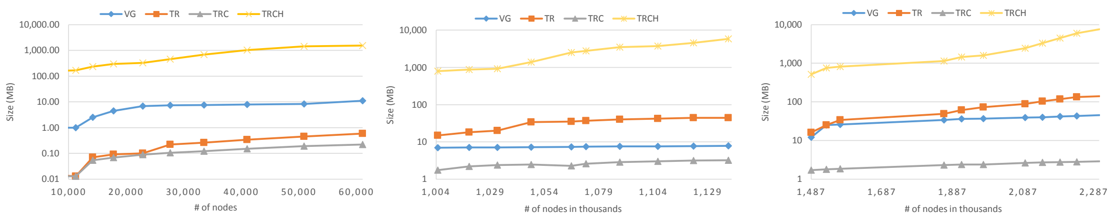


Figure 6: Size (log scale) for varying number of nodes in FB (left), YT (middle) and FL (right)

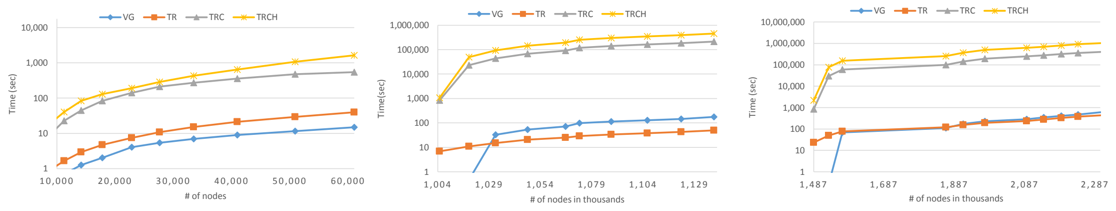


Figure 7: Construction time (log scale) for varying number of nodes in FB (left), YT (middle) and FL (right)

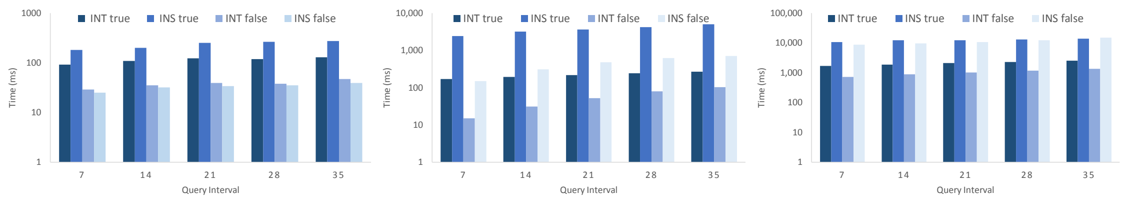


Figure 8: Query time (log scale) INS and INT for conjunctive queries in FB (left), YT (middle) and FL (right)

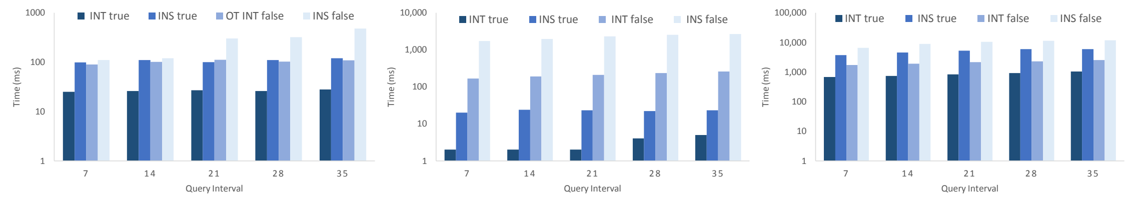


Figure 9: Query time (log scale) INS and INT for disjunctive queries in FB (left), YT (middle) and FL (right)

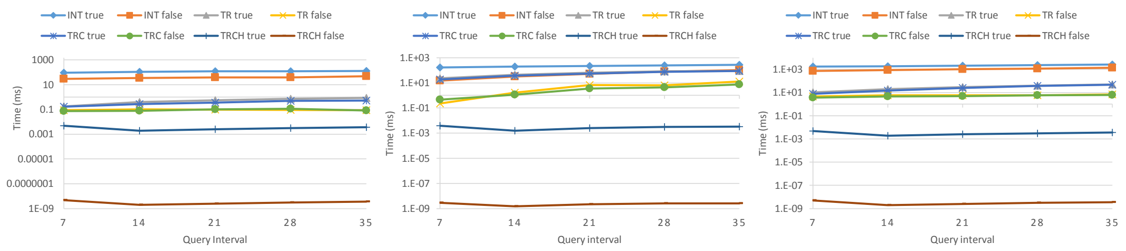


Figure 10: Query time (log scale) for conjunctive queries in FB (left), YT (middle) and FL (right)

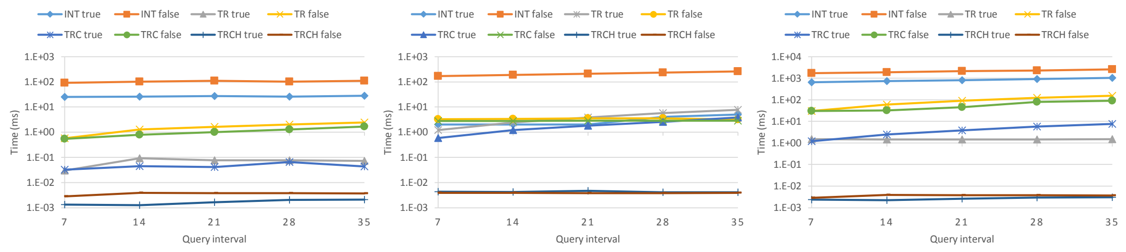


Figure 11: Query time (log scale) for disjunctive queries in FB (left), YT (middle) and FL (right)

9. REFERENCES

[1] Rakesh Agrawal, Alexander Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *SIGMOD*, pages 253–262, 1989.

[2] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In *WWW*, pages 237–248, 2014.

[3] Ramadhana Bramandia, Byron Choi, and Wee Keong Ng. On incremental maintenance of 2-hop labeling of graphs. In *WWW*, pages 845–854, 2008.

[4] Li Chen, Amarnath Gupta, and M. Erdem Kurul. Stack-based algorithms for pattern matching on dags. In *VLDB*, pages 493–504, 2005.

[5] Yangjun Chen and Yibin Chen. An efficient algorithm for answering graph reachability queries. In *ICDE*, pages 893–902, 2008.

[6] Jiefeng Cheng, Jeffrey Xu Yu, Xuemin Lin, Haixun Wang, and Philip S. Yu. Fast computing reachability labelings for large graphs with high compression rate. In *EDBT*, pages 193–204, 2008.

[7] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.*, 32(5):1338–1355, 2003.

[8] Ran Duan, Seth Pettie, and Hsin-Hao Su. Scaling algorithms for approximate and exact maximum weight matching. *Arxiv preprint arXiv:1112.0790*, 2011.

[9] Wenyu Huo and Vassilis J. Tsotras. Efficient temporal shortest path queries on evolving social graphs. In *Conference on Scientific and Statistical Database Management, SSDBM '14, Aalborg, Denmark, June 30 - July 02, 2014*, page 38, 2014.

[10] H. V. Jagadish. A compression technique to materialize transitive closure. *ACM Trans. Database Syst.*, 15(4):558–598, 1990.

[11] Christian S. Jensen and Richard T. Snodgrass. Temporal element. In *Encyclopedia of Database Systems*, page 2966. 2009.

[12] Ruoming Jin, Yang Xiang, Ning Ruan, and Haixun Wang. Efficiently answering reachability queries on very large directed graphs. In *SIGMOD*, pages 595–608, 2008.

[13] Udayan Khurana and Amol Deshpande. Efficient snapshot retrieval over historical graph data. In *ICDE*, pages 997–1008, 2013.

[14] Georgia Koloniari, Dimitris Souravlias, and Evaggelia Pitoura. On graph deltas for historical queries. *WOSS*, 2012.

[15] Ravi Kumar, Jasmine Novak, and Andrew Tomkins. Structure and evolution of online social networks. In *ACM SIGKDD*, pages 611–617, 2006.

[16] Alan G. Labouseur, Jeremy Birnbaum, Paul W. Olsen Jr, Sean R. Spillane, Jayadevan Vijayan, Jeong-Hyon Hwang, and Wook-Shin Han. The g* graph database: efficiently managing large distributed dynamic graphs. *Distributed and Parallel Databases*, To appear, 2014.

[17] Alan G. Labouseur, Paul W. Olsen, and Jeong-Hyon Hwang. Scalable and robust management of dynamic graph data. In *VLDB*, pages 43–48, 2013.

[18] Alan Mislove. Online social networks: Measurement, analysis, and applications to distributed information systems. Rice University, Department of Computer Science, 2009.

[19] Alan Mislove, Hema Swetha Koppula, Krishna P. Gummadi, and Bobby Bhattacharjee Peter Druschel. Growth of the flickr social network. In *ACM SIGCOMM WOSN*, pages 25–30, 2008.

[20] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and analysis of online social networks. In *ACM SIGCOMM IMC*, pages 29–42, 2007.

[21] Chenghui Ren, Eric Lo, Ben Kao, Xinjie Zhu, and Reynold Cheng. On querying historical evolving graph sequences. *PVLDB*, 4(11):726–737, 2011.

[22] Ralf Schenkel, Anja Theobald, and Gerhard Weikum. Hopi: An efficient connection index for complex xml document collections. In *EDBT*, pages 237–255, 2004.

[23] Ralf Schenkel, Anja Theobald, and Gerhard Weikum. Efficient creation and incremental maintenance of the hopi index for complex xml document collections. In *ICDE*, pages 360–371, 2005.

[24] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[25] Silke Trißl and Ulf Leser. Fast and practical indexing and querying of very large graphs. In *SIGMOD*, pages 845–856, 2007.

[26] Sebastiaan J. van Schaik and Oege de Moor. A memory efficient reachability data structure through bit vector compression. In *SIGMOD Conference*, pages 913–924, 2011.

[27] Bimal Viswanath, Alan Mislove, Meeyoung Cha, and Krishna P. Gummadi. On the evolution of user interaction in facebook. In *ACM SIGCOMM WOSN*, pages 37–42, 2009.

[28] Haixun Wang, Hao He, Jun Yang, Philip S. Yu, and Jeffrey Xu Yu. Dual labeling: Answering graph reachability queries in constant time. In *ICDE*, page 75, 2006.

[29] Yosuke Yano, Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths. In *CIKM*, pages 1601–1606, 2013.

[30] Hilmi Yildirim, Vineet Chaoji, and Mohammed J. Zaki. Grail: a scalable index for reachability queries in very large graphs. *VLDB J.*, 21(4):509–534, 2012.

[31] Hilmi Yildirim, Vineet Chaoji, and Mohammed J. Zaki. Dagger: A scalable index for reachability queries in large dynamic graphs. *CoRR*, abs/1301.0977, 2013.

Efficiently Computing Top-K Shortest Path Join

Lijun Chang[†], Xuemin Lin[†#], Lu Qin[¶], Jeffrey Xu Yu[‡], Jian Pei[§]

[†]*University of New South Wales, Australia, {ljchang,lxue}@cse.unsw.edu.au*
[#]*East China Normal University, China*
[¶]*University of Technology, Sydney, Australia, lu.qin@uts.edu.au*
[‡]*The Chinese University of Hong Kong, Hong Kong, China, yu@se.cuhk.edu.hk*
[§]*Simon Fraser University, Canada, jpei@cs.sfu.ca*

ABSTRACT

Driven by many applications, in this paper we study the problem of computing the top-k shortest paths from one set of target nodes to another set of target nodes in a graph, namely the top-k shortest path join (KPJ) between two sets of target nodes. While KPJ is an extension of the problem of computing the top-k shortest paths (KSP) between two target nodes, the existing technique by converting KPJ to KSP has several deficiencies in conducting the computation. To resolve these, we propose to use the best-first paradigm to recursively divide search subspaces into smaller subspaces, and to compute the shortest path in each of the subspaces in a prioritized order based on their lower bounds. Consequently, we only compute shortest paths in subspaces whose lower bounds are larger than the length of the current k-th shortest path. To improve the efficiency, we further propose an iteratively bounding approach to tightening lower bounds of subspaces. Moreover, we propose two index structures which can be used to reduce the exploration area of a graph dramatically; these greatly speed up the computation. Extensive performance studies based on real road networks demonstrate the scalability of our approaches and that our approaches outperform the existing approach by several orders of magnitude. Furthermore, our approaches can be immediately used to compute KSP. Our experiment also demonstrates that our techniques outperform the state-of-the-art algorithm for KSP by several orders of magnitude.

1. INTRODUCTION

Data are often modeled as graphs in many real applications such as social networks, information networks, gene networks, protein-protein interaction networks, and road networks. With the proliferation of graph data, significant research efforts have been made towards analyzing large graph data. These include the problem of computing the top-k shortest paths between two target nodes in a graph, namely the k shortest path (KSP) query.

KSP is a fundamental graph problem with many applications. In general, KSP is used in the applications that besides lengths, other constraints against the paths could not be precisely defined [12, 25]. For example, computing KSP between two sensitive accounts in a large social network enables end-users to identify all accounts involved in the top-k shortest paths [14]. In gene networks, the lengths of top-k shortest paths may be used to define the importance of a target gene to a source gene [26]. Other applications of KSP include multiple object tracking in pattern recognition [3], hypothesis generation in computational linguistics, and trip planning against road networks. Thus, KSP has been extensively studied [8, 9, 14, 15, 18, 24, 28].

The problem of computing the top-k shortest paths between two "conceptual" target nodes (instead of between two physical nodes) in a graph, called *the top-k shortest path join* (KPJ), is recently investigated in [15]. A conceptual node is a set of physical nodes in the graph, which can be identified by categories, concepts, and keywords in the above applications. While a KSP query is a special case of a KPJ query where each of the two conceptual target nodes only contains one physical node, KPJ can support more general application scenarios than KSP since a target node is allowed to be a set of physical nodes. For example, in a social network, the KPJ query can be used to detect user accounts involved in the top-k shortest paths between two criminal gangs to identify other "most suspicious" user accounts; the KPJ query can also be used in route planning where the destination is any one from a group of nodes (e.g., "IKEA"). In this paper, we study KPJ.

Motivations and Challenges. KPJ query shares similarity with but is different from the well-studied KSP query. To process a KPJ query, [15] reduces it to a KSP query by introducing a virtual target node for each conceptual target node and connecting every physical node in the conceptual node to it. Then, [15] proposes to use the state-of-the-art algorithm for KSP developed in [15]. Since the technique for solving KSP in [15] is based on the deviation paradigm [9, 28], applying KSP to solve KPJ, as proposed in [15], has the following two deficiencies. 1) Firstly, the deviation based techniques for KSP need to compute $O(k \cdot n)$ "candidate paths". The candidate paths are computed by iteratively extending all "prefixes" of the obtained l-th ($l < k$) shortest path, where n is the number of nodes in a graph, and each candidate path is computed by running an expensive shortest path algorithm; this is time-consuming. 2) Secondly, edges that are added to connect to a virtual target node for processing KPJ by using the KSP techniques depend on queries. This makes the existing index structures [7, 10] for computing shortest paths inapplicable. Thus, the candidate paths are computed by traversing the graph exhaustively; this is very costly.

Our Approaches. For presentation simplicity, in this paper we present our techniques against the simplified case, where one conceptual target node consists of one physical node only - called *source* node s, and the other conceptual target node may consist of multiple physical nodes - called *destination* nodes; then we extend our techniques to the general case where source nodes may also be multiple. Let $\mathcal{P}$ denote the set of all *simple* paths (i.e., paths without loops) from s to any of the destination nodes as the entire search space. Clearly, the result of KPJ is the set of k paths in $\mathcal{P}$ with the shortest lengths.

We adopt the best-first paradigm to recursively divide $\mathcal{P}$ into smaller subspaces, and then compute shortest paths for the generated subspaces in a prioritized order based on their lower bounds, where lower bound of a subspace is the lower bound of the length of all paths in the subspace. The top-k shortest paths may be iteratively obtained over the subspaces whose lower bounds are smaller than the length of the current k-th shortest path, while other subspaces can be safely pruned without the time-consuming shortest path computation.

We further propose to iteratively "guess" and tighten the lower bound τ of a subspace. Initially, we assign the value of τ as the length of the (1st) shortest path. Then, we always choose the subspace with the smallest τ to test whether the shortest path in it has length larger than $\alpha \cdot \tau$ (for an $\alpha > 1$). If the shortest path in the subspace can be determined larger than $\alpha \cdot \tau$, then we enlarge τ into $\alpha \cdot \tau$ for the subspace; otherwise the shortest path in the subspace is computed. Moreover, we propose two online-built index structures, $\mathsf{SPT_P}$ and $\mathsf{SPT_I}$, to significantly reduce the exploration area of a graph in the lower bound testing as briefly described above.

Contributions. Our primary contributions are summarized as follows.

- We propose a framework based on the best-first paradigm for processing KPJ queries which significantly reduces the number of shortest path computations.
- We propose an iteratively bounding approach to guessing and tightening the lower bounds, as well as two online-built index structures to speed-up the lower bound testing.
- We conduct extensive performance studies and demonstrate the scalability of our approaches which outperform the baseline approach [15] by several orders of magnitude.
- Moreover, our approaches can be immediately used to process KSP queries, and our experiments also demonstrate that our techniques outperform the state-of-the-art algorithm for KSP query by several orders of magnitude.

Organization. The rest of this paper is organized as follows. A brief overview of related work is given below. We give the preliminaries and problem statement in Section 2. The existing KSP-based approach is given in Section 3, in which we also discuss its deficiencies. We present the best-first paradigm in Section 4, in which we also implement a best-first approach. Under this paradigm, in Section 5 we propose an iteratively bounding approach and two online-built indexes for efficiently processing KPJ queries. In Section 6, we extend our techniques to cover other applications including the case that the source node has multiple physical nodes. We conducted extensive experimental studies and report our findings in Section 7, and we conclude this paper in Section 8.

Related Work. Given two nodes s and t in a graph G, the problem of computing the top-k shortest paths from s to t is a long-studied problem, which can be classified into two categories, 1) top-k simple shortest paths, and 2) top-k general shortest paths.

1) Top-k Simple Shortest Path. The existing algorithms for computing top-k simple shortest paths are based on the deviation paradigm proposed by Yen [9, 28], which has a time complexity of $O(k \cdot n \cdot (m + n \log n))$, where m is the number of edges in G and $O(m + n \log n)$ is the time complexity of computing single source shortest paths. Techniques to improve its efficiency in practice have been studied in [8, 14, 15, 18, 24], which shall be discussed in Section 3. We discuss using these techniques to process KPJ queries in Section 3.

2) Top-k General Shortest Path. Finding top-k general shortest paths is studied in [2, 12, 19], where cycles in paths are allowed. Since not enforcing paths to be simple, the top-k general shortest path problem is generally easier than its counterpart. Eppstein's algorithm [12] has the best time complexity, $O(m+n \log n+k)$, which is achieved by precomputing a shortest path tree rooted at the destination node and building a sophisticated data structure. Recently, the authors in [1] propose a heuristic search algorithm that has the same time complexity as [12]. However, due to different problem natures, these techniques are inapplicable to finding top-k simple shortest paths.

Finding Top-k Objects by Keywords. Finding k objects closest to a query location and containing user-given keywords has been studied in [13, 20, 22, 23, 29]. Ranking spatial objects by the combination of distance and relevance score is also studied in [4, 5, 27]. Distance oracles for node-label queries in a labeled graph are studied in [6, 17]; that is, given a query which contains a node and a label, it returns approximately the closest node to the query node that contains the query label. Nevertheless, the above queries are inherently different from KPJ, and their techniques cannot be applied to process KPJ queries.

2. PRELIMINARY

In this paper, we focus on a *weighted* and *directed graph* $G = (V, E, \omega)$, where V and E represent the set of nodes and the set of edges of G, respectively, and ω is a function assigning a weight to each edge in E. In G, nodes belong to categories, and each category represents a conceptual node consisting of all nodes belonging to that category. The number of nodes and the number of edges of G are denoted by $n = |V|$ and $m = |E|$, respectively.

A *path* P in G is a sequence of nodes $(v_1, \ldots, v_l)$ such that $(v_i, v_{i+1}) \in E, \forall 1 \leq i < l$, and we say that P consists of edges $(v_i, v_{i+1}), \forall 1 \leq i < l$. Here, v_1 and v_l are called the *source* node and *destination* node of P, respectively. P is a *simple* path if and only if all nodes in P are distinct (i.e., $v_i \neq v_j, \forall i \neq j$). A *prefix* of P is a subpath of P starting from the source node of P. The *length* of a path is defined as the total weight of its constituent edges; that is $\omega(P) = \sum_{(v_i,v_{i+1}) \in P} \omega(v_i, v_{i+1})$. The shortest distance from v_1 to v_2 is the shortest length among all paths from v_1 to v_2, denoted $\delta(v_1, v_2)$.

Definition 2.1: Given a category T, a path P is said to be a ***path to category*** T if its destination node is in V_T, where V_T is the set of nodes belonging to category T. □

Problem Statement: Given a graph G, we study the *top-k shortest path join* (KPJ) query, which aims at finding the top-k shortest simple paths $P_1, \ldots, P_k$ from a source node s to a category T (i.e., to any node in V_T).

Formally, a KPJ query is given as $Q = \{s, T, k\}$, where s is a source node in G, T represents a destination category, and k specifies the number of paths to find. It is to find k simple paths $P_1, \ldots, P_k$ such that: 1) each P_i is a path from s to category T; 2) $\omega(P_i) \leq \omega(P_{i+1}), \forall 1 \leq i < k$; 3) $\omega(P_k) \leq \omega(P)$ for any other path P from s to category T.

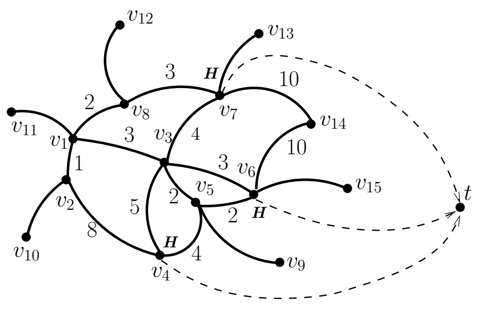


Figure 1: An example graph

Example 2.1: Fig. 1 illustrates a graph G, where $V = \{v_1, \ldots, v_{15}\}$ and nodes v_4, v_6, v_7 belong to category "H" (i.e., hotel). Here, edges are bidirectional, and weights are shown besides them with a default value 1. Consider a KPJ query $Q = \{v_1, \text{“}H\text{”}, 1\}$, which is to find the top-1 shortest path from v_1 to category "H". The top-1 path is $P_1 = (v_1, v_8, v_7)$ with $\omega(P_1) = 2 + 3 = 5$. □

For a KPJ query, V_T is the set of destination nodes. In the following, we assume that an inverted index [21] is offline built on the categories of nodes such that V_T can be efficiently retrieved online, and assume that a path is a simple path.

3. THE EXISTING KSP-BASED APPROACH

Reducing KPJ Query to KSP Query. The most related problem to KPJ is k shortest path (KSP) query defined below.

Definition 3.1:[28] Given a graph G, a KSP query $Q' = \{s, t, k\}$ is to find k simple paths $P_1, \ldots, P_k$ from s to t such that, 1) $\omega(P_i) \leq \omega(P_{i+1}), \forall 1 \leq i < k$, and 2) $\omega(P_k) \leq \omega(P)$ for any other path P from s to t. □

KSP query is a special case of KPJ query where V_T contains only one node. In other words, KSP query considers a single destination node while KPJ query considers multiple destination nodes. To process a KPJ query $Q = \{s, T, k\}$, [15] reduces it to a KSP query by adding a virtual destination node t to G and adding a directed edge from each node in V_T to t with a weight 0. Then, the result of Q on G is the same as the result of the KSP query $Q' = \{s, t, k\}$ on G_Q. Reconsider the KPJ query in Example 2.1, the modified graph G_Q is also shown in Fig. 1 with the virtual node t and the additional edges (dashed lines).

Deviation Algorithm (DA) for KSP Queries. The existing algorithms for KSP queries are based on the deviation paradigm [9, 28], denoted DA. It maintains a set C of candidate paths which include the next shortest path from s to t, and chooses k shortest paths from C one by one in a non-decreasing length order by incrementally updating C.

Pseudo-tree. The set of already chosen paths are encoded using a compact trie-like structure [21], called pseudo-tree. It is named because the same node may appear at several places in the tree; thus, we refer to nodes in a pseudo-tree as vertices to distinguish them from nodes in a graph. Let PT_i denote the pseudo-tree constructed for paths $P_1, \ldots, P_i$, and PT_0 consists of a single vertex s. PT_{i+1} is constructed by inserting P_{i+1} into PT_i by sharing the longest prefix; let d be the last vertex of the shared prefix, it is called the deviation vertex of P_{i+1} from PT_i. For example, Fig. 2 shows PT_1, PT_2, and PT_3, where PT_3 is constructed by inserting path (v_1, v_3, v_7, t) into PT_2, and v_3 is the deviation vertex.

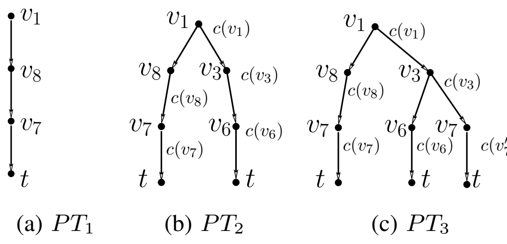


Figure 2: pseudo-trees

Candidate Path. Given a pseudo-tree PT_i, the DA algorithm maintains a set C_i of candidate paths, one corresponding to each vertex u in PT_i, denoted $c(u)$, which is *the shortest one among all paths from s to t that takes the path from s to u in PT_i as prefix and contains none of the outgoing edges of u in PT_i.* For example, in Fig. 2(c), $c(v_3)$ is the shortest one among all paths from s to t that take edge (v_1, v_3) as prefix and contain neither (v_3, v_6) nor (v_3, v_7); thus $c(v_3) = (v_1, v_3, v_5, v_6, t)$, and $C_3 = \{c(v_1), c(v_8), c(v_7), c(v_3), c(v_6), c(v_7')\}$.

Lemma 3.1: *[28]. Given a pseudo-tree PT_i and the corresponding C_i of candidate paths, the $(i+1)$-th shortest path from s to t is the path in C_i with shortest length.* □

Following from Lemma 3.1, the pseudocode of processing a KPJ query using DA is shown in Alg. 1, which is self-explanatory. The ingredient of Alg. 1 is to incrementally maintain PT_i and C_i after choosing each of the top-k paths.

```
Algorithm 1: DA($G_Q, Q' = \{s, t, k\}$)
1 Initialize $PT_0$ to contain a single vertex $s$;
2 Compute the shortest path $c(s)$ from $s$ to $t$, and $C_0 = \{c(s)\}$;
3 for each $i \leftarrow 1$ to $k$ do
4     $P_i \leftarrow$ the path in $C_{i-1}$ with the shortest length;
5     Construct $PT_i$ by inserting $P_i$ into $PT_{i-1}$, and let $d$ be the
      deviation vertex;
6     Construct $C_i$ by removing $P_i$ from $C_{i-1}$, computing the
      candidate paths corresponding to vertices in $P_i$ from $d$ to $t$, and
      inserting them into $C_{i-1}$;
7 return the $k$ paths $P_1, \ldots, P_k$;
```

Example 3.1: Fig. 2 demonstrates a running example for a KPJ query $Q = \{v_1, \text{“}H\text{''}, 3\}$ on the graph in Fig. 1. We first transform the graph G into G_Q, and reduce Q to a KSP query $Q' = \{v_1, t, 3\}$. The shortest path is $P_1 = (v_1, v_8, v_7, t)$ with length 5. After inserting P_1 into PT_0, the resulting PT_1 is shown in Fig. 2(a), where $C_1 = \{c(v_1), c(v_8), c(v_7)\}$. The 2nd shortest path is computed as $P_2 = c(v_1) = (v_1, v_3, v_6, t)$ which has the shortest length in C_1, and PT_2 is shown in Fig. 2(b). Then, candidate paths for v_1, v_3, v_6 are updated or computed, and $C_2 = \{c(v_8), c(v_7), c(v_1), c(v_3), c(v_6)\}$. The 3rd shortest path is $P_3 = c(v_3) = (v_1, v_3, v_7, t)$ with length 7. □

DA-SPT: Optimizations. The most time-consuming part of DA (Alg. 1) is Line 6, which needs to compute $O(k \cdot n)$ candidate paths in total. To efficiently compute a candidate path, several optimization techniques have been recently proposed [14, 24]. Pascoal [24] observes that, when computing $c(u)$ for u in a pseudo-tree PT, if the path formed by concatenating, 1) the path from s to u in PT, 2) an edge (u, v) in G_Q, and 3) the shortest path from v to t in G_Q, is simple, then it is $c(u)$. By preprocessing G_Q to generate a shortest path tree (SPT) storing shortest paths from all nodes to t, the path described above, if exists, can be found in constant time; otherwise, a shortest path algorithm is run to compute $c(u)$. Gao et al. [14, 15] improve Pascoal's approach by iteratively testing the above property during running Dijkstra's algorithm [11], and obtaining $c(u)$ once a simple path is found; this is known as the state-of-the-art approach, denoted DA-SPT, since a full SPT is built online.

Deficiencies of DA and DA-SPT. Both DA and DA-SPT are inefficient for processing KPJ queries due to the following three reasons. 1) Firstly, both need to compute $O(k \cdot n)$ candidate paths which are computed by iteratively extending all prefixes of the obtained l-th ($l < k$) shortest path; this is time-consuming. 2) Secondly, for processing a KPJ query using KSP techniques, the edges added to connect nodes in V_T to the virtual destination node depend on queries; this makes the existing index structures [7, 10] for efficiently computing shortest paths inapplicable. Thus, the candidate paths are computed by traversing the graph exhaustively, which is very costly. 3) Thirdly, although DA-SPT, compared to DA, can compute candidate paths more efficiently, it is time-consuming to construct the full SPT, which may be the dominating cost especially when the k shortest paths are short.

4. A BEST-FIRST APPROACH

In this section, to remedy the deficiencies of using the existing KSP techniques to process KPJ queries, we adopt a best-first paradigm which significantly reduces the number of shortest path computations thus enables fast query processing. In the following, we first discuss the paradigm, and then give an implementation of a best-first approach.

4.1 Best-First Paradigm

Given a KPJ query $Q = \{s, T, k\}$, let $\mathcal{P}_{s,T}(G)$ denote the set of all paths in G from s to category T (i.e., to any node in V_T). When the context is clear, $\mathcal{P}_{s,T}(G)$ is abbreviated to $\mathcal{P}$. Then, the query Q is to find the k paths in $\mathcal{P}$ with shortest lengths. Note that the size of $\mathcal{P}$ can be exponential to n.

Search Space and Subspace. The general idea is that we regard $\mathcal{P}$ as the entire search space $\mathcal{S}_0$. Then, the k paths in $\mathcal{P}$ with shortest lengths can be found by recursively dividing a subspace (initially $\mathcal{S}_0$) into smaller subspaces and computing the shortest path in each newly obtained subspace.

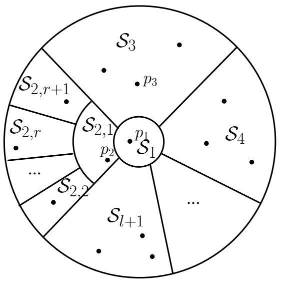


Figure 3: Overview of search space division

Before diving into the details, we first explain the main idea which is illustrated in Fig. 3. We conceptualize each path in $\mathcal{P}$ as a point, whose distance to the center (i.e., the origin) indicates the length of the path. Thus, the k paths with shortest lengths correspond to the k points closest to the center which can be computed as follows. First, we compute the closest point $P_1 = (v_1, \ldots, v_l)$ in the entire search space $\mathcal{S}_0 = \mathcal{P}$. Second, we divide $\mathcal{S}_0$ into $l+1$ subspaces, $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_l, \mathcal{S}_{l+1}$. Here, $\mathcal{S}_1$ consists of only P_1 and is excluded from further considerations. Each of the remaining subspaces, $\mathcal{S}_2, \ldots, \mathcal{S}_{l+1}$, represents the set of paths of $\mathcal{P}$ that share exactly the prefix of P_1 from v_1 to v_{i-1}; consequently, $\mathcal{S}_i \neq \mathcal{S}_j, \forall i \neq j$, and $\bigcup_{i=1}^{l+1} \mathcal{S}_i = \mathcal{S}_0$. Third, we compute the closest point in each of the l subspace, $\mathcal{S}_2, \ldots, \mathcal{S}_{l+1}$, and the one that is closest to the center among the l closest points represents the 2nd shortest path. Let it be $P_2 = (v'_1, \ldots, v'_r)$, and assume it is in $\mathcal{S}_2$. Fourth, we further divide $\mathcal{S}_2$ into $r+1$ subspaces, $\mathcal{S}_{2,1}, \mathcal{S}_{2,2}, \ldots, \mathcal{S}_{2,r}, \mathcal{S}_{2,r+1}$, and compute the closest point in each of them, where $\mathcal{S}_{2,1}$ consists of only P_2 and is excluded from further considerations. Thus, the point that is closest to the center among closest points in all subspaces $\mathcal{S}_3, \cdots, \mathcal{S}_{l+1}, \mathcal{S}_{2,2}, \ldots, \mathcal{S}_{2,r+1}$ represents the 3rd shortest path. We can repeat this process until k shortest paths are computed.

<u>Subspace Division.</u> We formally define a subspace below.

Definition 4.1: A ***subspace*** $\mathcal{S}$ is represented by a tuple $\langle P_{s,u}, X_u \rangle$, where $P_{s,u}$ is a path from s to u and X_u is a subset of the outgoing edges of u. It consists of all paths in $\mathcal{P}$ that *take* $P_{s,u}$ as prefix and *exclude* all edges of X_u. □

The entire search space $\mathcal{S}_0 (= \mathcal{P})$ is represented by $\langle P_{s,s} = (s), X_s = \emptyset \rangle$. Assume the shortest path in subspace $\langle P_{s,u}, X_u \rangle$ is P, then after choosing P as one of the k shortest paths, the subspace is divided into $l+1$ subspaces, where l is the number of nodes in the subpath of P from u to the destination node. The $l+1$ subspaces consist of a subspace containing only P, a subspace corresponding to node u (i.e., subspace $\langle P_{s,u}, X_u \cup \{(u,w)\} \rangle$), and one subspace corresponding to each node v in the subpath of P from u (exclusive) to the destination node (i.e., subspace $\langle P_{s,v}, \{(v,w')\} \rangle$), where $P_{s,v}$ is the prefix of P to v, and (u,w) and (v,w') are edges in P. It is important to note that the $l+1$ subspaces are disjoint, and their union is the original subspace $\langle P_{s,u}, X_u \rangle$ from which they are divided.

Example 4.1: Consider a KPJ query $Q = \{v_1, \text{“}H\text{”}, 2\}$ on the graph in Fig. 1. Initially, $\mathcal{S}_0 = \langle (v_1), \emptyset \rangle$ in which the shortest path is $P_1 = (v_1, v_8, v_7)$. Then, $\mathcal{S}_0$ is divided into four subspaces, $\mathcal{S}_1 = \{P\}$, $\mathcal{S}_2 = \langle (v_1), \{(v_1, v_8)\} \rangle$, $\mathcal{S}_3 = \langle (v_1, v_8), \{(v_8, v_7)\} \rangle$, and $\mathcal{S}_4 = \langle (v_1, v_8, v_7), \emptyset \rangle$, where $\mathcal{S}_1$ is the subspace containing only P_1. The 2nd shortest path is the one with shortest lengths among shortest paths in $\mathcal{S}_2, \mathcal{S}_3, \mathcal{S}_4$. □

Paradigm. It is easy to verify that there is a one-to-one correspondence between candidate paths defined in Section 3 and subspaces defined above. In the deviation paradigm, subspaces are implicitly maintained by storing candidate paths based on the fact that *each candidate path is the shortest path in a subspace*. Considering that shortest paths are expensive to compute, we remedy the deficiency of deviation paradigm by computing lower bounds of subspaces and pruning subspaces based on their lower bounds.

Definition 4.2: For a subspace $\mathcal{S} = \langle P_{s,u}, X_u \rangle$, we define the ***lower bound of a subspace***, denoted $\mathsf{lb}(\mathcal{S})$ (or $\mathsf{lb}(P_{s,u}, X_u)$), as the lower bound of lengths of all paths in $\mathcal{S}$, and denote the ***shortest path in a subspace*** by $\mathsf{sp}(\mathcal{S})$ (or $\mathsf{sp}(P_{s,u}, X_u)$). □

Based on lower bounds of subspaces, the best-first paradigm is shown in Alg. 2. Instead of directly computing the shortest path for each newly obtained subspace, we compute its lower bound first. All obtained subspaces and their lower bounds are maintained in a minimum priority queue $\mathcal{Q}$. Each entry of $\mathcal{Q}$ is a triple $\langle \mathcal{S}, \mathsf{lb}(\mathcal{S}), P \rangle$, where $\mathcal{S}$ and $\mathsf{lb}(\mathcal{S})$ are a subspace and its lower bound, respectively, and P is either $\emptyset$ or the shortest path in $\mathcal{S}$. Subspaces in $\mathcal{Q}$ are ranked by their lower bounds. Initially, $\mathcal{Q}$ contains a single entry representing the entire search space $\mathcal{S}_0$ (Line 1). Then, we iteratively remove the subspace with smallest lower bound from $\mathcal{Q}$, denoted $\langle \mathcal{S}, \mathsf{lb}(\mathcal{S}), P \rangle$ (Line 4): if $P \neq \emptyset$, then P is the

Algorithm 2: BestFirst$(G, Q = \{s, T, k\})$

```
1  Initialize a minimum priority queue $\mathcal{Q}$ to contain a single entry
   $\langle \mathcal{S}_0 = \langle (s), \emptyset \rangle, \mathsf{lb}(\mathcal{S}_0), \emptyset \rangle$;
2  $i \leftarrow 1$;
3  while $i \leq k$ do
4      $\langle \mathcal{S} = \langle P_{s,u}, X_u \rangle, \mathsf{lb}(\mathcal{S}), P \rangle \leftarrow$ remove the top entry from $\mathcal{Q}$;
5      if $P \neq \emptyset$ then
6          $P_i \leftarrow P$; $i \leftarrow i + 1$;
7          for each node $v$ in the subpath of $P$ from $u$ to the destination node do
8              Create a subspace $\mathcal{S}' = \langle P_{s,v}, X_v \rangle$;
9              $\mathsf{lb}(\mathcal{S}') \leftarrow \max\{\mathsf{CompLB}(P_{s,v}, X_v), \omega(P)\}$;
10             Put $\langle \mathcal{S}', \mathsf{lb}(\mathcal{S}'), \emptyset \rangle$ into $\mathcal{Q}$;
11     else
12         $\mathsf{sp}(\mathcal{S}) \leftarrow \mathsf{CompSP}(P_{s,u}, X_u)$;
13         if $\mathsf{sp}(\mathcal{S}) \neq \emptyset$ then Put $\langle \mathcal{S}, \omega(\mathsf{sp}(\mathcal{S})), \mathsf{sp}(\mathcal{S}) \rangle$ into $\mathcal{Q}$;
14 return the $k$ paths $P_1, \ldots, P_k$;
```

next shortest path to be output (Line 6), and we divide $\mathcal{S}$ by P and put those newly obtained subspaces into $\mathcal{Q}$ (Lines 7-10); otherwise, we compute the shortest path in $\mathcal{S}$ and put $\mathcal{S}$ into $\mathcal{Q}$ again with the computed shortest path (Lines 12-13). We will present an implementation of computing lower bound of a subspace and shortest path in a subspace in the next subsection.

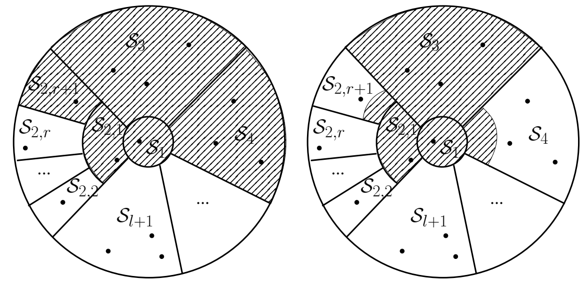
Lemma 4.1: *The set of shortest paths computed in Alg. 2 is a subset of that computed in Alg. 1; thus, the number of shortest path computations in Alg. 2 is not larger than that in Alg. 1. Assume that computing lower bound takes less time than computing shortest path for a subspace, then the time complexity of Alg. 2 is not larger than that of Alg. 1.* □

Proof Sketch: We prove the first part of Lemma 4.1 by proving that there is a one-to-one correspondence between subspaces inserted into $\mathcal{Q}$ in Alg. 2 and candidate paths computed in Alg. 1. This can be proved by induction. For $k = 1$, this is true, since there is only one subspace and one candidate path in Alg. 2 and Alg. 2, respectively. Now, we assume that this holds for general $k \geq 1$, then we prove that it also holds for $(k + 1)$. Since the k-th shortest path P_k corresponds to subspace $\mathcal{S}$ in Alg. 2, to obtain the $(k+1)$-th shortest path, we generate $O(n)$ new candidate paths from P_k in Alg. 1 and $O(n)$ new subspaces from $\mathcal{S}$ in Alg. 2; moreover, there is a one-to-one correspondence between these newly generated subspaces and newly generated candidate paths. Thus, there is a one-to-one correspondence between subspaces inserted into $\mathcal{Q}$ in Alg. 2 and candidate paths computed in Alg. 1. Considering that we compute shortest paths only for a subset of the subspaces inserted into $\mathcal{Q}$, the first part of the lemma holds.

Moreover, if we set all lower bounds computed at Line 9 of Alg. 2 to be 0, then the number of shortest path computations in Alg. 2 is the same as that in Alg. 1.

Second, in Alg. 2, any subspace is inserted into $\mathcal{Q}$ at most twice: once with computed lower bound (i.e., $P = \emptyset$), and once with computed shortest path. Thus, given that computing lower bound takes less time than computing shortest path for a subspace, the time complexity of Alg. 2 is not larger than that of Alg. 1. □

Following the proof of Lemma 4.1, we can also see that the maximum size of $\mathcal{Q}$ in Alg. 2 is $O(k \cdot n)$; moreover, given any algorithm computing a lower bound of a subspace, Alg. 2 correctly processes a KPJ query. Let P_k be the k-th shortest path for a KPJ query. Obviously, Alg. 2 does not compute shortest paths in subspaces whose lower bounds are larger than $\omega(P_k)$. Note that, in contrast, Alg. 1 needs to compute shortest paths in all subspaces in $\mathcal{Q}$. Conceptually, Fig. 4(a) shows by shadow the subspaces in which the shortest paths are computed: Alg. 2 computes only 5 shortest paths instead of $l + r + 2$ which is done by Alg. 1.



(a) Best First (b) Iteratively Bounding

Figure 4: Instances of shortest path computations

4.2 An Implementation of BestFirst

In the following, we present efficient techniques for computing a lower bound of a subspace (CompLB at Line 9 of Alg. 2) and for computing the shortest path in a subspace (CompSP at Line 12 of Alg. 2), denote the approach as BestFirst.

Algorithm 3: CompLB$(P_{s,u}, X_u)$

```
1  $lb \leftarrow +\infty$;
2  for each outgoing edge $(u, v)$ of $u$ do
3      if $v \notin P_{s,u}$ and $(u, v) \notin X_u$ then
4          Compute $\mathsf{lb}(v, V_T)$;
5          $lb \leftarrow \min\{lb, \omega(P_{s,u}) + \omega(u, v) + \mathsf{lb}(v, V_T)\}$;
6  return $lb$;
```

Computing Lower Bound of a Subspace. Given a subspace $\mathcal{S} = \langle P_{s,u}, X_u \rangle$, the set of paths in it corresponds to the set of paths from s to any node in V_T in a subgraph G' of G obtained as follows. We first remove all edges of X_u from G, and then for each node $v (\neq u)$ in $P_{s,u}$, we remove from G all outgoing edges of v except the one that is in $P_{s,u}$. Thus, for any two nodes u and v, the shortest distance from u to v in G' is lower bounded by that in G. Consequently, a naive lower bound of $\mathcal{S}$ is $\omega(P_{s,u}) + \mathsf{lb}(u, V_T)$, where $\mathsf{lb}(u, V_T)$ is the lower bound of shortest distance from u to any node in V_T, whose computation shall be discussed shortly. However, this is loose considering that many outgoing edges of u (i.e., X_u) are removed from G. Therefore, to estimate the lower bound of $\mathcal{S}$ (i.e., the shortest distance from s to any node in V_T in G') more accurately, we consider all valid outgoing edges (u, v) of u (i.e., $v \notin P_{s,u}$ and $(u, v) \notin X_u$), and choose the minimum estimation. The pseudocode is shown in Alg. 3 which is self-explanatory, and its correctness immediately follows from the above discussions.

Example 4.2: Continuing Example 4.1. After dividing $\mathcal{S}_0$ into $\mathcal{S}_1, \cdots, \mathcal{S}_4$, lower bounds of $\mathcal{S}_2, \mathcal{S}_3, \mathcal{S}_4$ are computed. Here, we illustrate how to compute $\mathsf{lb}(\mathcal{S}_2) = \mathsf{lb}((v_1), \{(v_1, v_8)\})$. v_1 has three valid outgoing edges, (v_1, v_2), (v_1, v_3), and (v_1, v_{11}), that can be considered for lower bound estimation. Thus, $\mathsf{lb}(\mathcal{S}_2)$ is computed as $\min\{\omega(v_1, v_2) + \mathsf{lb}(v_2, V_T), \omega(v_1, v_3) + \mathsf{lb}(v_3, V_T), \omega(v_1, v_{11}) + \mathsf{lb}(v_{11}, V_T)\}$. □

Computing $\mathsf{lb}(u, V_T)$. We propose a landmark-based approach [16] to estimating $\mathsf{lb}(u, V_T)$ in the following. Note that, the computation of $\mathsf{lb}(u, V_T)$ has not been studied in the literature.

A *landmark* is a subset of nodes, $L \subseteq V$. With L, the lower bound $\mathsf{lb}(u,v)$ of shortest distance from u to v is estimated as, $\mathsf{lb}(u,v) \doteq \max_{w\in L}\{\delta(w,v) - \delta(w,u)\}$, [1] where $\delta(w,v)$ and $\delta(w,u)$ are the shortest distance from w to v and to u, respectively, and are precomputed. This estimation is based on the fact that $\delta(w,u) + \delta(u,v) \geq \delta(w,v)$. Then, $\mathsf{lb}(u,V_T)$ can be estimated as,

$$\begin{aligned}\mathsf{lb}(u,V_T) &\doteq \min_{v\in V_T}\{\mathsf{lb}(u,v)\} \\ &= \min_{v\in V_T}\max_{w\in L}\{\delta(w,v) - \delta(w,u)\}\end{aligned} \quad (1)$$

However, the computation time is $O(|L|\cdot|V_T|)$ which is too costly especially when V_T is large. Therefore, motivated by the transformed graph G_Q in Section 3, we propose a new lower bound as follows,

$$\begin{aligned}\mathsf{lb}(u,V_T) &\doteq \max_{w\in L}\min_{v\in V_T}\{\delta(w,v) - \delta(w,u)\} \\ &= \max_{w\in L}\{\min\{\delta(w,v) \mid v\in V_T\} - \delta(w,u)\}\end{aligned} \quad (2)$$

The intuition is that, $\min\{\delta(w,v) \mid v \in V_T\}$ is the shortest distance from w to t in G_Q (i.e., $\delta(w,t)$); thus, Eq. (2) estimates $\mathsf{lb}(u,t)$. Consequently, $\mathsf{lb}(u,V_T)$ can be computed in $O(|L|)$ time by precomputing $\delta(w,t)$ for all $w \in L$ prior to any lower bound estimations. In what follows, we use Eq. (2) to compute $\mathsf{lb}(u,V_T)$.

Remarks & Time Complexity. Note that, the landmark index L is constructed offline in $O(|L|(m + n\log n))$ time where $O(m + n\log n)$ is the time complexity of a shortest path algorithm, while its space complexity is $O(|L|\cdot n)$. At the initialization phase of query processing, we compute $\delta(w,t)$ which is query dependent. The time complexity for computing $\delta(w,t)$ for all $w \in L$ is $O(|L|\cdot|V_T|)$; note that this is only computed once for each query.

Therefore, the time complexity of lower bound computation (i.e., Alg. 3) is $O(d(u)|L|)$ where $d(u)$ is the degree of u in G, since we traverse at most $d(u)$ edges at Line 2 while computing each $\mathsf{lb}(v,V_T)$ takes $O(|L|)$ time.

Computing Shortest Path in a Subspace. We use A* search [16] to compute $\mathsf{sp}(P_{s,u},X_u)$, denoted CompSP. In A* search, we consider only the valid edges (same as Line 3 of Alg. 3), and use Eq. (2) to estimate the shortest distance to destination. We omit the pseudocode.

Example 4.3: Continuing Example 4.2, after dividing $\mathcal{S}_0$, $\mathcal{Q}$ contains three subspaces, $\mathcal{S}_2, \mathcal{S}_3, \mathcal{S}_4$, together with their lower bounds. Assume the lower bounds are $\mathsf{lb}(\mathcal{S}_2) = 6$, $\mathsf{lb}(\mathcal{S}_3) = 11$, and $\mathsf{lb}(\mathcal{S}_4) = 12$. $\mathcal{S}_2$ has the smallest lower bound, and is removed from $\mathcal{Q}$. Then, $\mathsf{sp}(\mathcal{S}_2)$ is computed, which is the 2nd shortest path P_2, since its length is smaller than lower bounds of subspaces in $\mathcal{Q}$. Here, we compute the 2nd shortest path without computing shortest paths in subspaces $\mathcal{S}_3$ and $\mathcal{S}_4$. □

5. AN ITERATIVELY BOUNDING APPROACH

In this section, following the best-first paradigm in Section 4, we propose a new iteratively bounding approach to iteratively "guessing" and tightening lower bounds for subspaces in Section 5.1. Moreover, we propose two online-built indexes, in Section 5.2 and Section 5.3, respectively, based on which we can reduce the exploration area of a graph dramatically in tightening lower bounds.

5.1 Iteratively Bounding

In BestFirst, we prune subspaces whose lower bounds are larger than $\omega(P_k)$, where P_k is the k-th shortest path for a KPJ query. Therefore, BestFirst will run fast if we can prune more subspaces based on their lower bounds (i.e., by computing tighter/larger lower bounds for subspaces), considering that computing shortest path is time-consuming. However, in general, computing a tighter lower bound takes longer time, and in the extreme case computing the shortest path in a subspace provides the tightest lower bound. In Alg. 3, we present a light-weight lower bound estimation by considering only the immediate neighbors of u. Intuitively, we can compute a tighter lower bound by exploring multi-hop neighbors of u (e.g., neighbors of neighbors).

We propose to guess and tighten lower bounds of subspaces in a controlled manner by a threshold τ, which is achieved by a procedure TestLB. In a nutshell, given a subspace $\mathcal{S}$ and a threshold τ, TestLB tests whether the shortest path in $\mathcal{S}$ has a length larger than τ: if it is, then the lower bound is set as τ; otherwise, the shortest path $\mathsf{sp}(\mathcal{S})$ is obtained and returned. Details of TestLB will be discussed shortly. Ideally, we can set τ as $\omega(P_k)$, then all subspaces whose lower bounds are larger than $\omega(P_k)$ are pruned with the least amount of effort; however, P_k is unknown. Considering that TestLB takes longer time for a larger τ, we iteratively enlarge τ, and the k shortest paths of a KPJ query will be found once τ becomes no smaller than $\omega(P_k)$.

Algorithm 4: IterBound$(G, Q = \{s,T,k\})$
```
1  Compute the shortest path $P'$ from $s$ to any node in $V_T$ in $G$;
2  Initialize a minimum priority queue $\mathcal{Q}$ to contain a single entry
   $\langle \mathcal{S}_0 = \langle (s), \emptyset\rangle, \omega(P'), P'\rangle$;
3  $i \leftarrow 1$; $\tau \leftarrow \omega(P')$;
4  while $i \leq k$ do
5      $\langle \mathcal{S} = \langle P_{s,u}, X_u\rangle, \mathsf{lb}(\mathcal{S}), P\rangle \leftarrow$ remove the top entry from $\mathcal{Q}$;
6      if $P \neq \emptyset$ then
7          Same as Lines 6-10 of Alg. 3;   /* $P_i \leftarrow P$, $i \leftarrow i+1$,
           and divide $\mathcal{S}$ into subspaces */;
8      else
9          $\tau \leftarrow \alpha\cdot\max\{\mathsf{lb}(\mathcal{S}), \mathcal{Q}.top().key\}$; /* Enlarge $\tau$ */;
10         $P \leftarrow$ TestLB$(P_{s,u}, X_u, \tau)$;
11         if $P \neq \emptyset$ then Put $\langle \mathcal{S}, \omega(P), P\rangle$ into $\mathcal{Q}$;
12         else Put $\langle \mathcal{S}, \tau, \emptyset\rangle$ into $\mathcal{Q}$;
13 return the $k$ paths $P_1, \ldots, P_k$;
```

The algorithm IterBound is given in Alg. 4, which is similar to Alg. 2. We first compute the shortest path P' in G (Line 1), put subspace $\mathcal{S}_0 = \langle (s), \emptyset\rangle$ with path P' into $\mathcal{Q}$ (Line 2), and initialize τ as $\omega(P')$ (Line 3). Then, we iteratively remove the subspace $\mathcal{S}$ with smallest lower bound, together with path P, from $\mathcal{Q}$ (Line 5). If P is not empty, then it is the next shortest path P_i, and we perform the same subspace division as Lines 6-10 of Alg. 2. Otherwise, we enlarge τ (Line 9), test whether the shortest path of $\mathcal{S}$ has a distance larger than τ (Line 10), and put $\mathcal{S}$ back into $\mathcal{Q}$ together with either the computed shortest path P or a larger lower bound τ depending on what TestLB returns (Lines 11-12).

Note that τ controls the computation of a tighter lower bound which we want to make larger but a larger τ will make TestLB slow. In our approach, we use a parameter α to control the speed of increasing τ iteratively. Here, α can be any real number larger than 1, and we use $\alpha = 1.1$ as default. At Line 9, we enlarge τ as $\alpha\cdot\max\{\mathsf{lb}(\mathcal{S}), \mathcal{Q}.top().key\}$, where $\mathcal{Q}.top().key$ is the key value (i.e., lower bound) in the top entry of $\mathcal{Q}$ and is defined to be $+\infty$ if $\mathcal{Q} = \emptyset$. The intuition is that, $\omega(P_k)$ should be not much larger than $\omega(P_1)$, the initial τ (Line 3). Note that $\mathsf{lb}(\mathcal{S})$ is the previous τ we have tested for $\mathcal{S}$; thus, the lower bound τ we tested for a subspace increases by a factor of at least α. Therefore, we iteratively enlarge τ from $\omega(P_1)$ to approach $\omega(P_k)$, and obtain the k shortest paths for a KPJ query once τ becomes no smaller than $\omega(P_k)$.

[1]Note that this triangle inequality holds for the shortest distances based on any distance metrics not only Euclidean distance. Thus, our techniques work for general graphs.

Theorem 5.1: *Given* TestLB*,* IterBound *correctly computes k shortest paths for a* KPJ *query.* □

Proof Sketch: IterBound follows the best-first paradigm of Alg. 2, except that we iteratively compute lower bounds of subspaces. Moreover, it is easy to prove that the lower bound τ computed for the same subspace $\mathcal{S}$ at Line 9 is strictly increasing (assuming that $\alpha > 1$). Therefore, let P_i be the correct i-th shortest path, we can prove that τ will be no less than $\omega(P_i)$ when the algorithm terminates, and once τ becomes no less than $\omega(P_i)$, the path P_i will be computed and inserted into $\mathcal{Q}$, thus will be output. □

IterBound acts the same as BestFirst if we set τ as $+\infty$. Nevertheless, by iteratively enlarging τ with an initial value $\omega(P_1)$, IterBound runs much faster than BestFirst by pruning more subspaces. Conceptually, Fig. 4(b) shows the subspaces in which the shortest paths are computed by IterBound. Compared to Fig. 4(a), $\mathcal{S}_4$ and $\mathcal{S}_{2,r+1}$ are pruned based on their tighter lower bounds that are computed by TestLB. The shadow areas of $\mathcal{S}_4$ and $\mathcal{S}_{2,r+1}$ in Fig. 4(b) indicate the exploration areas of TestLB in testing lower bounds.

Testing Lower Bound. TestLB tests whether the shortest path in a subspace has length larger than a given threshold τ. This is achieved by considering multi-hop neighbors of u, denoted V'. Given τ, V' are those nodes v with $\delta_{\mathcal{S}}(s,v)+\mathsf{lb}(v,V_T) \leq \tau$, where $\delta_{\mathcal{S}}(s,v)$ is the shortest distance from s to v constrained in $\mathcal{S}$ (i.e., G' defined in Section 4.2). After obtaining V', if $\mathsf{V}' \cap V_T = \emptyset$ then $\omega(\mathsf{sp}(\mathcal{S})) > \tau$, otherwise, $\mathsf{sp}(\mathcal{S})$ is obtained by backtracking from $\mathsf{V}' \cap V_T$.

Algorithm 5: TestLB($P_{s,u}, X_u, \tau$)

```
1  Initialize a minimum-priority queue $\mathcal{Q}_V$ to contain $\langle u, 0\rangle$;
2  $d_s(u) \leftarrow \omega(P_{s,u})$, and $d_s(v) \leftarrow +\infty$ for all other nodes;
3  while $\mathcal{Q}_V \neq \emptyset$ do
4      $v \leftarrow$ remove the top node from $\mathcal{Q}_V$;
5      if $v \in V_T$ then return the path formed by concatenating $P_{s,u}$
       with the computed shortest path from $u$ to $v$;
6      else for each outgoing edge $(v,w)$ of $v$ do
7          if $w \notin P_{s,u}, (v,w) \notin X_u$ and $d_s(v) + \omega(v,w) < d_s(w)$
           then
8              $d_s(w) \leftarrow d_s(v) + \omega(v,w)$;
9              Compute $\mathsf{lb}(w, V_T)$;
10             if $d_s(w) + \mathsf{lb}(w,V_T) \leq \tau$ then
11                 Put $\langle w, d_s(w) + \mathsf{lb}(w,V_T)\rangle$ into $\mathcal{Q}_V$;
12 return $\emptyset$;
```

The pseudocode of TestLB is shown in Alg. 5, which is similar to A* search [16], where $d_s(v)$ stores the length of a path from s to v constrained in $\mathcal{S}$. We maintain the explored nodes together with their estimated distances (i.e., node v with $d_s(v) + \mathsf{lb}(v,V_T)$) in a minimum-priority queue $\mathcal{Q}_V$, which is initialized to contain u; nodes in $\mathcal{Q}_V$ are ranked by their estimated distances. Then, nodes are iteratively removed from $\mathcal{Q}_V$ (Line 4), and their neighbors are inserted into $\mathcal{Q}_V$ (Lines 6-11), until $\mathcal{Q}_V = \emptyset$ or we get a node from V_T; the latter case implies that $\mathsf{sp}(\mathcal{S})$ has been computed (Line 5). Here, V' is the set of nodes removed from $\mathcal{Q}_V$. Note that, following from [16], when a node v is removed from $\mathcal{Q}_V$, $d_s(v)$ stores the shortest distance from s to v constrained in $\mathcal{S}$ (i.e., $\delta_{\mathcal{S}}(s,v)$), and each node is removed from $\mathcal{Q}_V$ at most once.

The efficiency of TestLB is due to that, we only put into $\mathcal{Q}_V$ those nodes whose estimated distance are not larger than τ, as ensured by Line 10, which prunes a lot of nodes especially for a small τ.

Lemma 5.1: *Given a subspace $\mathcal{S}$ and τ,* TestLB *returns* $\mathsf{sp}(\mathcal{S})$ *if* $\omega(\mathsf{sp}(\mathcal{S})) \leq \tau$*, and returns $\emptyset$ otherwise.* □

Proof Sketch: First of all, we remark that if we remove Lines 9-10 from Alg. 5, then it is the same as the A* search algorithm [16] that computes the shortest path in $\mathcal{S}$. Thus, if $\omega(\mathsf{sp}(\mathcal{S})) \leq \tau$, then TestLB returns $\mathsf{sp}(\mathcal{S})$, since every node that is pruned at Line 10 will not be in $\mathsf{sp}(\mathcal{S})$ due to the nature of lower bound.

Secondly, we prove that if $\omega(\mathsf{sp}(\mathcal{S})) > \tau$, then TestLB returns $\emptyset$. The reason is that, for every node v obtained at Line 4 of Alg. 5, we have $d_s(v) \leq \tau$ due to the pruning at Line 10. Thus, Alg. 5 cannot find the path $\mathsf{sp}(\mathcal{S})$, and returns $\emptyset$. □

Time Complexity. The time complexity of Alg. 5 is $O(m'+n' \log n')$, where n' and m' are the number of visited nodes and edges in Alg. 5 (specifically, at Line 6), respectively. In the worst case, $n' = n$ and $m' = m$; thus the time complexity is $O(m + n \log n)$. However, in practice, n' and m' are usually small and much smaller than n and m, respectively.

Example 5.1: First, let's consider TestLB($(v_1, v_3), \{(v_3, v_6)\}, 6$). v_3 has three valid out-neighbors, v_4, v_5, v_7. v_4 and v_7 are pruned because $d_{v_1}(v_3) + \omega(v_3,v_4) > 6$ and $d_{v_1}(v_3) + \omega(v_3,v_7) > 6$. Assume $\mathsf{lb}(v_5, V_T) = 2$, then v_5 is also pruned since $d_{v_1}(v_3) + \omega(v_3,v_5)+\mathsf{lb}(v_5,V_T) = 7$. Thus, TestLB($(v_1,v_3), \{(v_3,v_6)\}, 6$) returns $\emptyset$. Now, let's consider $\tau = 7$. Among the three valid out-neighbors, v_4 is pruned because $d_{v_1}(v_3) + \omega(v_3,v_4) = 8$; v_5 and v_7 are put into $\mathcal{Q}_V$ with lower bounds 7. Then, v_5 is removed from $\mathcal{Q}_V$, and its out-neighbor, v_6, is put into $\mathcal{Q}_V$ with lower bound 7. After that, either v_6 or v_7 is removed from $\mathcal{Q}_V$, and the shortest path in $\langle (v_1, v_3), \{(v_3, v_6)\}\rangle$ is obtained. □

5.2 Partial Shortest Path Tree

Motivated by DA-SPT, in this subsection we propose to compute and store a partial SPT, denoted $\mathsf{SPT_P}$, which provides a more accurate estimation of $\mathsf{lb}(v,V_T)$. Recall that $\mathsf{lb}(v,V_T)$ is used in TestLB to prune nodes, thus a more accurate estimation of TestLB will result in faster computation time. In contrast to DA-SPT which online constructs a full SPT by incurring high overheads, we obtain $\mathsf{SPT_P}$ as a by-product of computing the shortest path in G (i.e., Line 1 of Alg. 4) without any extra cost.

Algorithm 6: PartialSPT(G, s, T)

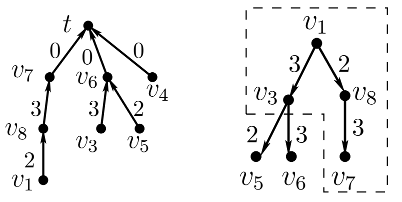
```
1  Initialize an empty minimum-priority queue $\mathcal{Q}_T$;
2  $\mathsf{SPT_P} \leftarrow$ a virtual root node $t$;        /* Build SPT_P */;
3  for each $w \in V_T$ do
4      Put $\langle w, \mathsf{lb}(s,w)\rangle$ into $\mathcal{Q}_T$, $d_t(w) \leftarrow 0, p(w) \leftarrow t$;
5  while $\mathcal{Q}_T \neq \emptyset$ do
6      $v \leftarrow$ remove the top node from $\mathcal{Q}_T$;
7      Add $v$ as a child of $p(v)$ to $\mathsf{SPT_P}$;  /* Build SPT_P */;
8      if $v = s$ then return the path from $s$ to $t$;
9      for each incoming edge $(w,v)$ of $v$ do
10         if $d_t(v) + \omega(w,v) < d_t(w)$ then
11             $d_t(w) \leftarrow d_t(v) + \omega(w,v), p(w) \leftarrow v$;
12             Put $\langle w, d_t(w) + \mathsf{lb}(s,w)\rangle$ into $\mathcal{Q}_T$;
```

The algorithm to construct $\mathsf{SPT_P}$ is given in Alg. 6, denoted PartialSPT, which is the A* search algorithm for computing the shortest path from s to any node in V_T in G by adding Lines 2,7. The algorithm runs in the reverse graph of G, since we want to compute shortest paths from different nodes to any node in V_T. $\mathcal{Q}_T$ is similar to $\mathcal{Q}_V$ in Alg. 5 and initially contains all nodes of V_T (Lines 3-4). Then, nodes are iteratively removed from $\mathcal{Q}_T$ (Line 6) and their incoming edges are explored (Lines 9-12). The shortest path from s to any node in V_T is obtained when s is removed from

$\mathcal{Q}_T$ (Line 8). For all nodes v removed from $\mathcal{Q}_T$, we add v as a child of $p(v)$ to $\mathsf{SPT_P}$. Intuitively, $\mathsf{SPT_P}$ contains all nodes removed from $\mathcal{Q}_T$ prior to s when computing the shortest path from s to any node in V_T.

Proposition 5.1: *For nodes $v \in \mathsf{SPT_P}$, the path from v to t in $\mathsf{SPT_P}$ is the shortest path from v to any node in V_T in G.* □

Computing $\mathsf{lb}(v, V_T)$ using $\mathsf{SPT_P}$. Both CompLB and TestLB, which are invoked by IterBound, require computing $\mathsf{lb}(v, V_T)$ for any $v \in V$. Eq. (2) computes $\mathsf{lb}(v, V_T)$ using a landmark-based approach. By utilizing $\mathsf{SPT_P}$, we can compute a more accurate $\mathsf{lb}(v, V_T)$ as follows. If v is in $\mathsf{SPT_P}$, then $\mathsf{lb}(v, V_T)$ is computed as the length of the path from v to t in $\mathsf{SPT_P}$, the correctness of which directly follows from Proposition 5.1; otherwise, it is computed by Eq. (2). Here, we give $\mathsf{SPT_P}$ a higher priority, because if $v \in \mathsf{SPT_P}$, then the lower bound computed using $\mathsf{SPT_P}$ is guaranteed to be not smaller than that by Eq. (2); for lower bound, the larger the better.



(a) $\mathsf{SPT_P}$ (b) $\mathsf{SPT_I}$

Figure 5: Partial and incremental SPT

Example 5.2: Fig. 5(a) shows the $\mathsf{SPT_P}$ constructed for $Q = \{v_1, \text{“}H\text{”}, 3\}$. For each node v in $\mathsf{SPT_P}$, its distance to t in $\mathsf{SPT_P}$ is the shortest distance from v to any node in V_T and can be used as an estimation of $\mathsf{lb}(v, V_T)$. For example, $\mathsf{lb}(v_3, V_T)$ is estimated as 3. □

IterBound-$\mathsf{SPT_P}$ Approach. We denote the approach that uses $\mathsf{SPT_P}$ to estimate $\mathsf{lb}(v, V_T)$ in Alg. 4 as IterBound-$\mathsf{SPT_P}$. The correctness of IterBound-$\mathsf{SPT_P}$ directly follows from that of IterBound and the above discussions.

5.3 Incremental Shortest Path Tree

$\mathsf{SPT_P}$ includes all nodes of V_T which can be large for a KPJ query, thus may take long time to construct. In this subsection, we propose an incremental SPT, denoted $\mathsf{SPT_I}$, by pruning nodes in V_T that are far-away from the source node s. Moreover, we incrementally enlarge $\mathsf{SPT_I}$, based on which we identify a new property for reducing the exploration area of a graph by TestLB.

Constructing $\mathsf{SPT_I}$. To prune from $\mathsf{SPT_I}$ those nodes in V_T that are far-away from s, in $\mathsf{SPT_I}$ we compute and store shortest paths from s to each node of a subset of V. Recall that, in $\mathsf{SPT_P}$, we store shortest paths from each node of a subset of V to V_T. Thus, the construction of $\mathsf{SPT_I}$ is run on G by starting from s, and consists of two phases. In the first phase, we construct an initial $\mathsf{SPT_I}$, which is a by-product of computing the shortest path from s to any node in V_T in a similar fashion to PartialSPT (i.e., Alg. 6) while running on G and starting from s; this is invoked at Line 1 of Alg. 4. In the second phase, we incrementally enlarge $\mathsf{SPT_I}$ by IncrementalSPT, which is invoked after Line 9 and before Line 10 of Alg. 4.

The pseudocode of IncrementalSPT is shown in Alg. 7, which is self-explanatory. The general idea is to include into $\mathsf{SPT_I}$ all nodes of V that are on paths from s to any node in V_T whose lengths are not larger than τ. Therefore, IncrementalSPT iteratively removes

Algorithm 7: IncrementalSPT(G, T, τ)

```
1 while $\mathcal{Q}_T.top().key \le \tau$ do
2     $v \leftarrow$ remove the top node from $\mathcal{Q}_T$;
3     Add $v$ as a child of $p(v)$ to $\mathsf{SPT_I}$;
4     if $v \in V_T$ then Add $v$ to $D$;
5     for each outgoing edge $(v, w)$ of $v$ do
6         if $d_s(v) + \omega(v, w) < d_s(w)$ then
7             $d_s(w) \leftarrow d_s(v) + \omega(v, w), p(w) \leftarrow v$;
8             Put $\langle w, d_s(w) + \mathsf{lb}(w, V_T)\rangle$ into $\mathcal{Q}_T$;
```

the top node v from $\mathcal{Q}_T$ (Line 2), and adds v into $\mathsf{SPT_I}$ (Line 3). Meanwhile, the subset of V_T that are in $\mathsf{SPT_I}$ is maintained into a set D (Line 4), which will be used later to improve the performance of testing lower bound for a subspace. In Fig. 5(b), the subtree in the rectangle shows the initial $\mathsf{SPT_I}$ constructed, and the entire tree is the resulting $\mathsf{SPT_I}$ for $\tau = 7$. Here, $D = \{v_6, v_7\}$. $\mathsf{SPT_I}$ has the following property.

Proposition 5.2: *The $\mathsf{SPT_I}$ constructed by Alg. 7 contains all nodes on paths from s to any node in V_T whose lengths are no larger than τ.* □

Algorithm 8: CompLB-$\mathsf{SPT_I}$ $(P_{t,u}, X_u)$

```
1 if $u \ne t$ then $N(u) \leftarrow$ {in-neighbors of $u$} else $N(u) \leftarrow D$;
2 $lb \leftarrow +\infty$;
3 for each $v \in N(u)$ do
4     if $v \notin P_{t,u}$ and $(v, u) \notin X_u$ then
5         if $v \notin \mathsf{SPT_I}$ then Compute $\mathsf{lb}(s, v)$ using Eq. (2);
6         else $\mathsf{lb}(s, v) \leftarrow$ the distance from $s$ to $v$ in $\mathsf{SPT_I}$;
7         $lb \leftarrow \min\{lb, \omega(P_{t,u}) + \omega(v, u) + \mathsf{lb}(s, v)\}$;
8 if $u = t$ and $lb = +\infty$ and $D \ne V_T$ then $lb \leftarrow 0$;
9 return $lb$;
```

Computing Initial Lower Bound for a Subspace using $\mathsf{SPT_I}$. We compute the lower bound of a subspace using $\mathsf{SPT_I}$ in a similar way to CompLB (Alg. 3), by considering the immediate neighbors of u. The algorithm is shown in Alg. 8, denoted CompLB-$\mathsf{SPT_I}$. Note that, here $P_{t,u}$ is a path from u to t in G and X_u is a subset of the incoming edges to u. CompLB-$\mathsf{SPT_I}$ differs from CompLB in the following two aspects. Firstly, $\mathsf{lb}(s, v)$ is estimated by utilizing $\mathsf{SPT_I}$ in the same way as that in Section 5.2. Secondly, when $u = t$, instead of considering all nodes of V_T, we consider only the subset that is in $\mathsf{SPT_I}$ (i.e. D). The reason is that, the entire V_T can be very large, while the small subset D is sufficient for our purpose of computing $\mathsf{lb}(P_{t,u}, X_u)$, which saves a lot of computations.

The correctness of CompLB-$\mathsf{SPT_I}$ when $u \ne t$ directly follows from that of CompLB. However, when $u = t$, there are two cases depending on whether there are any valid incoming edges to u. 1) If there is no valid incoming edge to u (i.e., every node of $N(u)$ is either in $P_{t,u}$ or in X_u, Lines 3-4), then lb will be $+\infty$. We reassign lb to 0 if $D \ne V_T$. 2) Otherwise, $lb \ne +\infty$, which is guaranteed to be a lower bound of the subspace $\langle P_{t,u}, X_u\rangle$.

Testing Lower Bound using $\mathsf{SPT_I}$. By utilizing $\mathsf{SPT_I}$, we propose a more efficient algorithm for testing lower bound, denoted TestLB-$\mathsf{SPT_I}$. We modify TestLB to develop TestLB-$\mathsf{SPT_I}$ in the same fashion as the development of CompLB-$\mathsf{SPT_I}$. Moreover, *we prune all nodes that are not in $\mathsf{SPT_I}$ from consideration* (i.e., from putting into $\mathcal{Q}_V$). Consequently, every $\mathsf{lb}(s, w)$ is computed as the distance from s to w in $\mathsf{SPT_I}$; that is, Eq. (2) is not evaluated. Therefore, TestLB-$\mathsf{SPT_I}$ is more efficient than TestLB. We prove the correctness of $\mathsf{TestLB\text{-}SPT_I}$ in the following lemma.

Lemma 5.2: *Given a subspace $\mathcal{S}$ and τ,* $\mathsf{TestLB\text{-}SPT_I}$ *returns $\emptyset$ if $\omega(\mathsf{sp}(\mathcal{S})) \geq \tau$, and returns* $\mathsf{sp}(\mathcal{S})$ *otherwise.* □

Proof Sketch: The lemma follows from Lemma 5.1 and Proposition 5.2. □

Example 5.3: We show running $\mathsf{TestLB\text{-}SPT_I}$ for subspace $\langle(v_7), \{(v_7, v_8)\}\rangle$ with $\tau = 6$, where $\mathsf{SPT_I}$ is shown in Fig. 5(b). Among v_7's in-neighbors, only v_3 is considered, since v_{13} and v_{14} are not in $\mathsf{SPT_I}$. For v_3, $\mathsf{lb}(v_1, v_3) = 3$ which is the distance in $\mathsf{SPT_I}$. Then, v_3 is also pruned since $\omega(v_3, v_7) + \mathsf{lb}(v_1, v_3) = 7$, and $\mathsf{TestLB\text{-}SPT_I}$ returns $\emptyset$. For $\tau = 7$, $\mathsf{SPT_I}$ remains the same. Then, v_3 is not pruned and the shortest path in $\langle(v_7), \{(v_7, v_8)\}\rangle$ is obtained, which is (v_1, v_3, v_7) with length 7. □

$\mathsf{IterBound\text{-}SPT_I}$ **Approach.** Based on $\mathsf{SPT_I}$ and the discussions above, we propose an approach following Alg. 4 for processing KPJ queries, denoted $\mathsf{IterBound\text{-}SPT_I}$. It runs on the reverse graph of G, and a subspace is represented by $\langle P_{t,u}, X_u\rangle$ where X_u is a subset of the incoming edges to u.

$\mathsf{IterBound\text{-}SPT_I}$ improves the efficiency by computing $\mathsf{SPT_I}$ and pruning all nodes not in $\mathsf{SPT_I}$ from consideration when conducting the iteratively bounding search. That is, we take as input only the small subgraph of G induced by nodes in $\mathsf{SPT_I}$. Note that, $\mathsf{SPT_I}$ enlarges for a larger τ, and the subgraph induced by nodes in $\mathsf{SPT_I}$ also enlarges; this guarantees that we can correctly process any KPJ query.

Time Complexity. The time complexity of the $\mathsf{IterBound\text{-}SPT_I}$ approach is $O(kn(m' + n' \log n'))$, where n' and m' are the number of nodes and edges, respectively, in the subgraph G' of G that is induced by nodes w with $d_s(w) + lb(w, V_T) <= \tau$ (see Line 10 of Alg. 5) for the largest τ obtained in $\mathsf{IterBound\text{-}SPT_I}$. Note that, n' and m' are usually small in real applications; thus, $\mathsf{IterBound\text{-}SPT_I}$ runs much faster than DA (see Alg. 1).

6. EXTENSIONS

In the following, we extend our techniques to other applications including the case that the source node also has multiple physical nodes and the case that landmarks are not available.

General KPJ. A general KPJ (GKPJ) query is an extension of KPJ query where the source node also has multiple physical nodes, denote as $Q = \{S, T, k\}$, where both S and T are categories. It is to compute the top-k shortest paths from any node in V_S to any node in V_T, where V_S is the set of nodes of V belonging to category S. We can convert a GKPJ query to a KPJ query by introducing a virtual source node s and connecting s to all nodes in V_S with weights 0. Then, Q is reduced to a KPJ query $Q' = \{s, T, k\}$ on the new graph, and all our proposed techniques can be used to process the query.

Computing without Landmark. Our techniques are presented based on landmarks which are used to estimate $\mathsf{lb}(u, V_T)$. Nevertheless, when landmarks are not available, all our techniques can still be directly applied by setting all $\mathsf{lb}(u, V_T)$ (i.e., computed by Eq. (2)) to be 0. Specifically, for $\mathsf{IterBound\text{-}SPT_I}$, the landmark is only used for constructing the $\mathsf{SPT_I}$ using A* search [16], as discussed in Section 5.3; thus, without landmark, we construct the $\mathsf{SPT_I}$ by setting $\mathsf{lb}(u, V_T)$ to be 0 (the A* search then becomes the Dijkstra's algorithm [11]), while other parts of $\mathsf{IterBound\text{-}SPT_I}$ remain the same.

Moreover, without landmark, our techniques still perform well; the reasons are as follows. The $\mathsf{IterBound\text{-}SPT_I}$ approach mainly consists of two parts: 1) incrementally constructing the partial shortest path tree $\mathsf{SPT_I}$, 2) computing lower bound or shortest path for a subspace. The dominating cost comes from the second part, while landmark is only used in the first part. Thus, by running $\mathsf{IterBound\text{-}SPT_I}$ without landmark will only increase the cost of the first part which is not a big factor in the total cost.

7. PERFORMANCE STUDIES

We conduct extensive performance studies to evaluate the efficiency of our approaches against the baseline approaches for processing KPJ queries. The following algorithms are implemented:

- DA [28] and DA-SPT [15]. They are in the deviation paradigm as discussed in Section 3, and are the baseline approaches for processing KPJ queries.
- BestFirst. It is the best-first approach as discussed in Section 4.
- IterBound, $\mathsf{IterBound_P}$, and $\mathsf{IterBound_I}$. They are the iteratively bounding approaches with or without SPT as discussed in Section 5. $\mathsf{IterBound_P}$ and $\mathsf{IterBound_I}$ are abbreviations of $\mathsf{IterBound\text{-}SPT_P}$ and $\mathsf{IterBound\text{-}SPT_I}$, respectively.
- $\mathsf{IterBound_I}$-NL. It is the $\mathsf{IterBound_I}$ approach however without landmark, as discussed in Section 6.

Note that, 1) in our testings, a KSP query is also considered as a KPJ query where the query category uniquely identifies the destination node; 2) the baseline approaches for processing KPJ queries are the state-of-the-art techniques for processing KSP queries.

All algorithms are implemented in C++ and compiled with GNU GCC by -O3 option. All tests are conducted on a PC with an Intel(R) Xeon(R) 2.66GHz CPU and 4GB memory running Linux. We evaluate the performance of the algorithms on real datasets as follows.

Dataset	#Nodes	#Edges
CAL	106,337	213,964
SJ	18,263	47,594
SF	174,956	443,604
COL	435,666	1,042,400
FLA	1,070,376	2,687,902
USA	6,262,104	15,119,284

Table 1: Summary of dataset

Datasets. We use six real road networks with real/synthetic points of interest (POIs), and each POI belongs to a category. They are: California road network (CAL), San Joaquin road network (SJ), San Francisco road network (SF), Colorado road network (COL), Florida road network (FLA), and Western USA road network (USA). The first three are downloaded from `www.cs.utah.edu/~lifeifei/SpatialDataset.htm`, and the last three are from DIMACS (`www.dis.uniroma1.it/~challenge9/download.shtml`). A summary is given in Table 1.

POIs. The CAL dataset is provided with real POIs, which have 62 different categories. For the other five road networks, we generate synthetic POIs randomly located on nodes. For each road network, we generate four sets of POIs, denoted T_1, T_2, T_3, T_4, corresponding to different number of physical destination nodes (i.e., $n \times 10^{-4}, 5n \times 10^{-4}, 10n \times 10^{-4}, 15n \times 10^{-4}$ POIs, respectively), where n is the number of nodes in a road network. For example, for

COL, $|T_1| = 43$, $|T_2| = 217$, $|T_3| = 435$, and $|T_4| = 653$. Note that, we generate the POIs in such a way that $T_1 \subset T_2 \subset T_3 \subset T_4$.

Graphs. We model each road network with POIs as a graph G. Here, an edge (u, v) in G represents a road segment, and has a non-negative weight $\omega(u, v)$ which can be any measure of the road segment, such as distance, travel time, travel cost, and etc. We take distance as weight in our experiments. Each node belongs to the categories of POIs that are located on it.[2]

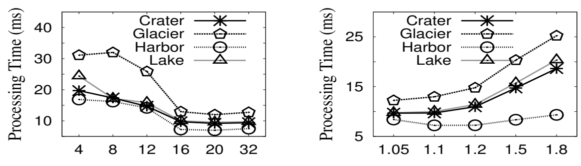
Queries. A query consists of a source node s, a destination node set V_T indicated by category T, and a value k indicating the number of paths to found. For each query, we first choose a category T, and then randomly generate source nodes. For the CAL dataset, we consider four representative categories, *"Glacier"*, *"Lake"*, *"Crater"*, and *"Harbor"*, which have 1, 8, 14, and 94 physical nodes, respectively. For the other datasets, we consider T_1, T_2, T_3, and T_4, and choose T_2 by default.

For a destination category T, the source nodes in a query are randomly generated as follows. We sort all nodes in increasing order regarding their shortest path lengths to category T, partition them into 5 groups, and generate 5 query sets, Q_1, Q_2, Q_3, Q_4, Q_5, each of which consists of 100 nodes randomly selected from the corresponding group. Thus, nodes in Q_i are closer to destination nodes than nodes in Q_j do, for $i < j$. We use Q_3 as the default query set.

k is chosen from 10, 20, 30, and 50, with 20 as the default.

7.1 Experimental Results

Eval-I: Parameters. We evaluate the influence of landmark size $|L|$ and parameter α on the performance of $\mathsf{IterBound_I}$.



(a) Varying $|L|$ (b) Varying α

Figure 6: Parameter testing on CAL ($Q_3, k = 20$)

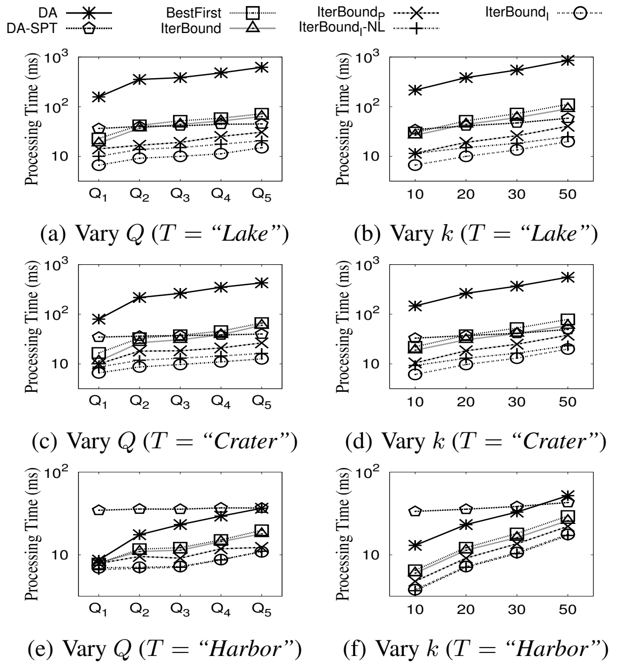
Choosing $|L|$. In our approaches, we use landmarks for estimating $\mathsf{lb}(v, V_T)$, the lower bound of shortest distance from v to any node in V_T. The landmarks are chosen following the most popular way in [16].[3] Fig. 6(a) shows the processing time of $\mathsf{IterBound_I}$ for different $|L|$ values. Clearly, when $|L|$ increases from 4 to 16, the processing time decreases, because more landmarks can provide more accurate estimation of $\mathsf{lb}(v, V_T)$. However, when $|L|$ increases from 16 to 32, the processing time increases a little due to longer computation time of $\mathsf{lb}(v, V_T)$. Therefore, we choose $|L| = 16$.

Choosing α. The running time of $\mathsf{IterBound_I}$ for different α values are illustrated in Fig. 6(b). Recall that α is used in our iterative bounding approaches for controlling the increasing ratio of τ (i.e., controlling the computation of a tighter lower bound, see Alg. 4). The running time increases when α increases from 1.1 to 1.8 due to building a larger $\mathsf{SPT_I}$. However, when α decreases from 1.1 to 1.05, the processing time also increases due to taking more iterations to reach the final τ. Therefore, we choose $\alpha = 1.1$.

Note that: 1) among our parameters, τ is determined by α; 2) better choices of $|L|$ and α will improve the performance of our algorithm marginally as shown in Fig. 6. It will be our future work to automatically find the best choice of $|L|$ and α.

[2]For simplicity, we assume that POIs are located on the nodes of G. When a POI is on an edge (u, v), we can add a new node w to G and connect w with u and v to replace (u, v). Note that, given a query with category T, we only need to consider the set of POIs belonging to category T.

[3]We firstly pick a random start node and select the farthest node from the start node as the first landmark, and then iteratively choose the node that is farthest away from the current set of landmarks as the next landmark.



(a) Vary Q (T = *"Lake"*) (b) Vary k (T = *"Lake"*)

(c) Vary Q (T = *"Crater"*) (d) Vary k (T = *"Crater"*)

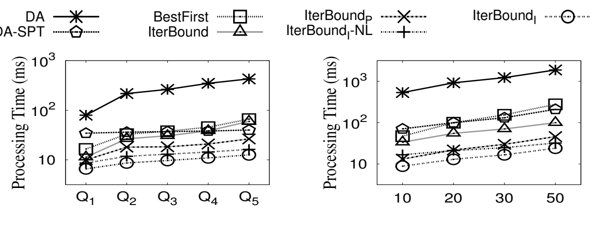
(e) Vary Q (T = *"Harbor"*) (f) Vary k (T = *"Harbor"*)

Figure 7: Against baseline approaches on CAL (Varying Q, k)

Eval-II: Against the Baseline Approaches. Here, we evaluate the performances of our approaches against the baseline approaches on CAL dataset.

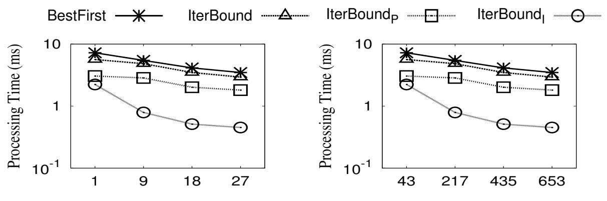
KPJ *Query.* The processing time of the seven approaches by varying query sets and k is demonstrated in Fig. 7, where the destination category is chosen from *"Lake"*, *"Crater"*, and *"Harbor"*. In general, all our approaches, BestFirst, IterBound, $\mathsf{IterBound_P}$, $\mathsf{IterBound_I}$, and $\mathsf{IterBound_I}$-NL, outperform the two baseline approaches, DA and DA-SPT. This is because our approaches use a best-first paradigm to reduce the number of shortest path computations. In Figures 7(a)-7(d), DA-SPT outperforms DA because DA-SPT online builds a full SPT to facilitate the shortest path computation. However, in Fig. 7(e)-7(f), DA-SPT performs worse due to the dominating cost of building the full SPT. When the lengths of shortest paths increase (i.e., varying Q from Q_1 to Q_5), the running time of all approaches increases except DA-SPT which is steady, due to the dominating cost of constructing the full SPT. Moreover, although without landmarks, $\mathsf{IterBound_I}$-NL outperforms all other approaches except $\mathsf{IterBound_I}$ across all testings. The trend of the processing time of these approaches by varying k is similar to that of varying query set. One exception is that, the processing time of DA-SPT also slightly increases due to computing more shortest paths for larger k.

KSP *Query.* We test the approaches for processing KSP queries by setting the destination category as *"Glacier"* which has only one



(a) Vary Q (T = *"Glacier"*) (b) Vary k (T = *"Glacier"*)

Figure 8: Testing KSP queries on CAL (Varying Q and k)



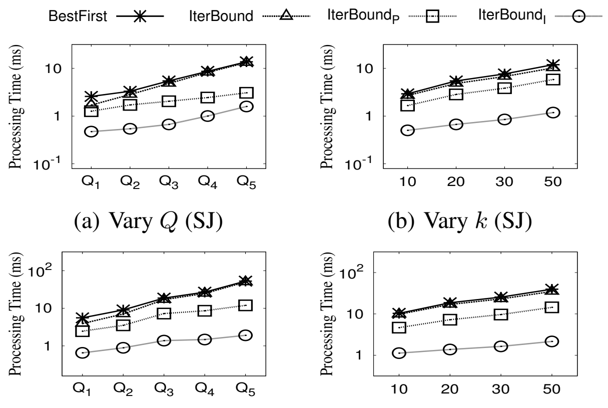
(a) SJ (b) COL

Figure 10: Vary #(destination nodes) ($Q = Q_3$, $k = 20$)

physical destination node; thus, the KPJ query is a KSP query. The results are shown in Fig. 8, which are similar to that for KPJ queries in Fig. 7.

Summary. There is no clear winner between DA and DA-SPT, and all our approaches perform better than these two baseline approaches. Despite using the same techniques, IterBound$_l$ outperforms IterBound$_l$-NL, which demonstrates the effectiveness of using landmarks for estimating lower bounds. Therefore, in the following testings, we omit DA, DA-SPT, and IterBound$_l$-NL, and evaluate the other approaches which use different techniques and all use landmark.

Eval-III: Evaluating Our Approaches. In this testing, we evaluate the efficiency of our different approaches, BestFirst, IterBound, IterBound$_P$, and IterBound$_l$, on SJ and COL.



(a) Vary Q (SJ) (b) Vary k (SJ)

(c) Vary Q (COL) (d) Vary k (COL)

Figure 9: Our approaches by varying Q and k ($T = T_2$)

Varying Q and k. The running time of the approaches on SJ and COL by varying Q and k is shown in Fig. 9. Similar to that in Fig. 7, the running time of these four approaches increases when either Q varies from Q_1 to Q_5 or k increases. IterBound slightly outperforms BestFirst due to less number of shortest path computations, however with more expensive lower bound computations. IterBound$_P$ performs better than IterBound because of the faster lower bound testing. IterBound$_l$ runs faster than IterBound$_P$ because IterBound$_l$ can further reduce the exploration area of a graph by SPT$_l$.

Varying Number of Destination Nodes ($|T|$). Fig. 10 shows the processing time of these four approaches on SJ and COL by varying the number of destination nodes (i.e., varying $|T|$). For all these four approaches, the processing time decreases, when the number of destination nodes increases (i.e., T varies from T_1 to T_4). This is because the shortest paths become shorter for more number of destination nodes as shown in Fig. 11 which will be discussed shortly. IterBound$_l$ outperforms IterBound$_P$ which then outperforms BestFirst and IterBound. The improvement of IterBound$_l$ over IterBound$_P$ becomes more significant when there are more destination nodes, since IterBound$_l$ can prune destination nodes and reduce the exploration area of a graph by SPT$_l$.

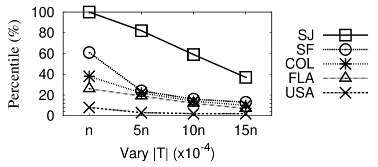
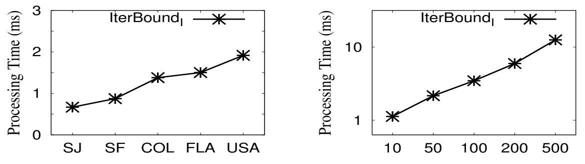


Figure 11: Shortest path length (Varying #(destination nodes))

Fig. 11 illustrates the influence of the number of destination nodes on the shortest path lengths. Specifically, for each category T_i, we compute the longest length of shortest paths from nodes to T_i, and report its percentile position in the observations of all $n \cdot n$ shortest path lengths in the graph. For all datasets, the shortest path lengths decrease with more number of destination nodes; thus all approaches run faster as shown in Fig. 10. Note that, for a specific T_i, the number of destination nodes belonging to T_i are different, thus the shortest path lengths vary for different datasets; for example, for T_1, the number of destination nodes for SJ, SF, COL, FLA, USA are 1, 17, 43, 107, and 626, respectively.

Summary. IterBound$_l$ outperforms the other approaches, BestFirst, IterBound, and IterBound$_P$, across all different datasets, different number of destination nodes, different Q, and different k. Thus, in the following we only evaluate our IterBound$_l$ approach.

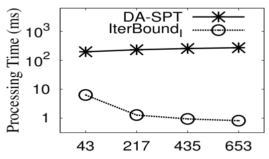


(a) Vary graph size ($k = 20$) (b) Vary k (COL)

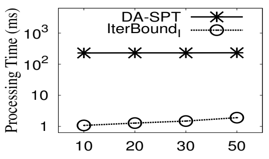
Figure 12: Scalability of IterBound$_l$ ($T = T_2$, $Q = Q_3$)

Eval-IV: Scalability Testing. The scalability testing results of IterBound$_l$ by varying graph size and k are shown in Fig. 12. Although the running time increases when either the graph size or k increases, IterBound$_l$ is scalable enough to process very large graphs. For example, when the graph size increases 40 times (i.e., from SJ to USA), the running time of IterBound$_l$ only increases slightly (e.g., by no more than 3 times).

Eval-V: GKPJ Testing. In this evaluation, we test the efficiency of IterBound$_l$ over DA-SPT, the state-of-the-art approach, for GKPJ



(a) Vary #(destination nodes) ($k = 20$)



(b) Vary k ($T = T_2$)

Figure 13: GKPJ query (COL)

queries $Q = \{S, T, k\}$. Here, the source category S has 4 physical nodes which are randomly chosen. Fig. 13 shows the running time by varying the number of destination nodes (i.e., $|T|$) or k. The trends of running time of DA-SPT and IterBound$_l$ are similar to the previous evaluations. The improvement of IterBound$_l$ over DA-SPT is more significant (e.g., by two orders of magnitude). This is because the lengths of k shortest paths become smaller with multiple source nodes.

8. CONCLUSION

In this paper, we studied the problem of top-k shortest path join (KPJ). We adopted the best-first paradigm to reduce the number of shortest path computations, compared to the existing deviation paradigm, by pruning subspaces based on their lower bounds. To improve the efficiency, we further proposed an iteratively bounding approach to tightening lower bounds of subspaces which is achieved by lower bound testing. Moreover, we proposed index structures to significantly reduce the exploration area of a graph in lower bound testing. We conducted extensive performance studies using real road networks, and demonstrated that our proposed approaches significantly outperform the baseline approaches for KPJ queries. Furthermore, our approaches can be immediately used to process KSP queries, and they also outperform the state-of-the-art algorithm for KSP queries by several orders of magnitude.

Acknowledgements. Lijun Chang is supported by ARC DE150100563. Xuemin Lin is supported by NSFC61232006, ARC DP120104168, ARC DP140103578, and ARC DP150102728. Lu Qin is supported by ARC DE140100999. Jeffrey Xu Yu is supported by Research Grants Council of the Hong Kong SAR, China, 14209314 and 418512. Jian Pei is supported in part by an NSERC Discovery grant.

9. REFERENCES

[1] H. Aljazzar and S. Leue. K*: A heuristic search algorithm for finding the k shortest paths. *Artif. Intell.*, 175(18):2129–2154, 2011.

[2] R. Bellman and R. Kalaba. On kth best policies. *Journal of the Society for Industrial and Applied Mathematics*, 1960.

[3] J. Berclaz, F. Fleuret, E. Türetken, and P. Fua. Multiple object tracking using k-shortest paths optimization. *IEEE Trans. Pattern Anal. Mach. Intell.*, 33(9), 2011.

[4] X. Cao, G. Cong, and C. S. Jensen. Retrieving top-k prestige-based relevant spatial web objects. *PVLDB*, 3(1), 2010.

[5] X. Cao, G. Cong, C. S. Jensen, and B. C. Ooi. Collective spatial keyword querying. In *Proc. of SIGMOD'11*, 2011.

[6] S. Chechik. Improved distance oracles for vertex-labeled graphs. *CoRR*, abs/1109.3114, 2011. informal publication.

[7] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.*, 32(5), 2003.

[8] E. de Queirós Vieira Martins and M. M. B. Pascoal. A new implementation of yen's ranking loopless paths algorithm. *4OR*, 1(2), 2003.

[9] E. de QueirÍős Vieira Martins, M. M. B. Pascoal, and J. L. E. dos Santos. The k shortest loopless paths problem, 1998.

[10] D. Delling, A. V. Goldberg, R. Savchenko, and R. F. Werneck. Hub labels: Theory and practice. In *Proc. of SEA'14*, 2014.

[11] E. W. Dijkstra. A note on two problems in connexion with graphs. *NUMERISCHE MATHEMATIK*, 1(1), 1959.

[12] D. Eppstein. Finding the k shortest paths. *SIAM J. Comput.*, 28(2), 1998.

[13] I. D. Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In *Proc. of ICDE'08*, pages 656–665, 2008.

[14] J. Gao, H. Qiu, X. Jiang, T. Wang, and D. Yang. Fast top-k simple shortest paths discovery in graphs. In *Proc. of CIKM'10*, 2010.

[15] J. Gao, J. X. Yu, H. Qiu, X. Jiang, T. Wang, and D. Yang. Holistic top-k simple shortest path join in graphs. *IEEE Trans. Knowl. Data Eng.*, 24(4):665–677, 2012.

[16] A. V. Goldberg and C. Harrelson. Computing the shortest path: $A*$ search meets graph theory. In *Proc. of SODA'05*, 2005.

[17] D. Hermelin, A. Levy, O. Weimann, and R. Yuster. Distance oracles for vertex-labeled graphs. In *Proc. of ICALP'11*, 2011.

[18] J. Hershberger, M. Maxel, and S. Suri. Finding the *k* shortest simple paths: A new algorithm and its implementation. *ACM Transactions on Algorithms*, 3(4), 2007.

[19] W. Hoffman and R. Pavley. A method for the solution of the nth best path problem. *J. ACM*, 6, October 1959.

[20] C. S. Jensen, J. Kolárvr, T. B. Pedersen, and I. Timko. Nearest neighbor queries in road networks. In *Proc. of GIS'03*, 2003.

[21] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[22] M. R. Kolahdouzan and C. Shahabi. Voronoi-based k nearest neighbor search for spatial network databases. In *Proc. of VLDB'04*, 2004.

[23] S. Nutanong and H. Samet. Memory-efficient algorithms for spatial network queries. In *Proc. of ICDE'13*, 2013.

[24] M. M. B. Pascoal. Implementations and empirical comparison of *k* shortest loopless path algorithms, Nov. 2006.

[25] D. Quercia, R. Schifanella, and L. M. Aiello. The shortest path to happiness: Recommending beautiful, quiet, and happy routes in the city. In *Proc. of Hypertext'14*, 2014.

[26] Y.-K. Shih and S. Parthasarathy. A single source *k*-shortest paths algorithm to infer regulatory pathways in a gene network. *Bioinformatics*, 28(12), 2012.

[27] P. Venetis, H. Gonzalez, C. S. Jensen, and A. Y. Halevy. Hyper-local, directions-based ranking of places. *PVLDB*, 4(5), 2011.

[28] J. Y. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17, 1971.

[29] C. Zhang, Y. Zhang, W. Zhang, and X. Lin. Inverted linear quadtree: Efficient top k spatial keyword search. In *Proc. of ICDE'13*, 2013.

SIEF: Efficiently Answering Distance Queries for Failure Prone Graphs

Yongrui Qin
School of Computer Science
The University of Adelaide
Adelaide, SA 5005, Australia
yongrui.qin@adelaide.edu.au

Quan Z. Sheng
School of Computer Science
The University of Adelaide
Adelaide, SA 5005, Australia
michael.sheng@adelaide.edu.au

Wei Emma Zhang
School of Computer Science
The University of Adelaide
Adelaide, SA 5005, Australia
wei.zhang01@adelaide.edu.au

ABSTRACT

Shortest path computation is one of the most fundamental operations for managing and analyzing graphs. A number of methods have been proposed to answer shortest path distance queries on static graphs. Unfortunately, there is little work on answering distance queries on *dynamic* graphs, particularly graphs with edge failures. Today's real-world graphs, such as the social network graphs and web graphs, are evolving all the time and link failures occur due to various factors, such as people stopping following others on Twitter or web links becoming invalid. Therefore, it is of great importance to handle distance queries on these failure-prone graphs. This is not only a problem far more difficult than that of static graphs but also important for processing distance queries on evolving or unstable networks. In this paper, we focus on the problem of computing the shortest path distance on graphs subject to edge failures. We propose SIEF, a Supplemental Index for Edge Failures on a graph, which is based on distance labeling. Together with the original index created for the original graph, SIEF can support distance queries with edge failures efficiently. By exploiting properties of distance labeling on static graphs, we are able to compute very compact distance labeling for all singe-edge failure cases on dynamic graphs. We extensively evaluate our algorithms using six real-world graphs and confirm the effectiveness and efficiency of our approach.

Categories and Subject Descriptors

E.1 [**Data**]: Data Structures—*Graphs and networks*; H.2.8 [**Database management**]: Database Applications—*Graph Indexing and Querying*

General Terms

Algorithms, Experimentation, Performance

Keywords

Shortest paths, distance query, 2-hop labeling, edge failure

1. INTRODUCTION

Recent years have witnessed the fast emergence of massive graph data in many application domains, such as the World Wide Web, linked data technology, online social networks, and Web of Things [21, 19, 22, 25]. In a graph, one of the most fundamental challenges centers on the efficient computation of the shortest path or distance between any given pair of vertices. For instance, distances or the numbers of links between web pages on a web graph can be considered a robust measure of web page relevancy, especially in relevance feedback analysis in web search [21]. In RDF graphs of linked data, the shortest path distance from one entity to another is important for ranking entity relationships and keyword querying [19, 14]. For online social networks, the shortest path distance can be used to measure the closeness centrality between users [22].

A large body of indexing techniques have been recently proposed to process exact shortest path distance queries on graphs [10, 23, 9, 8, 2, 26, 15]. Among them, a significant portion of indexes are based on *2-hop* distance labeling, which is originally proposed by Cohen et al. [12]. The 2-hop distance labeling techniques pre-compute a label for each vertex so that the shortest path distance between any two vertices can be computed by giving their labels only. These labeling indexes, such as [10, 8, 2, 15], have been proved to be efficient, i.e., being able to answer a distance query within microseconds.

Motivation. The above mentioned approaches generally make the assumption that graphs are static. However, in reality, many graphs are subject to edge failures. In this paper, we refer to graphs that are not subject to edge failures as *stable graphs*, i.e., static graphs. Similarly, we refer to graphs that are subject to edge failures as *unstable graphs*. For example, the emerging social Web of Things calls for graph data management with edge failures because smart things are normally moving and their connectivity could be intermittent, leading to frequent and unpredictable changes in the corresponding graph models [11, 25]. Another example is web graphs. It is not uncommon that some web links become invalid as the web evolves. All these are examples of unstable graphs, which are common in the real-world, calling for efficient graph computations by considering link failures. We believe that it is imperative to design novel algorithms that can compute shortest path indexes for fast response on distance queries avoiding any failed edge. Some real-world applications/scenarios that require the computa-

10.5441/002/edbt.2015.14

tion of shortest path distance avoiding a failed edge are described in the following.

Scenario 1. *The* **most vital arc** *problem [17, 6] aims to identify the edge on a given shortest path and the removal of this edge results in the longest replacement path. Here, a replacement path means a shortest path from a source vertex to a destination vertex in a graph that avoids a specified edge. To find the most vital arc in a graph, we need to compute the shortest path distances efficiently when we are given an arc (i.e., an edge) to avoid.*

Scenario 2. *In the* **sensitivity analysis** *and in many analytical applications of transportation networks, government agencies need to evaluate different road segments (i.e., to find how much a road segment is worth) through Vickrey pricing [16], such that maintenance budget can be allocated accordingly, or the amount of tolls can be adjusted reasonably [24]. For example, if tolls are not charged appropriately and avoiding an expensive toll point causes only a small detour, then it is more likely that most drivers would take the detour, rather than pay for the toll.*

Scenario 3. *In order to develop* **game-theoretic and price-based mechanisms** *to share bandwidth and other network resources, a natural economic question is [16]: how much is an edge in a network worth to a user who wants to send data between two nodes along a shortest path? Or in other words, what is the penalty of avoiding an edge in the given network?*

These application scenarios reveal an urge for handling shortest path computations in a graph with single-edge failures. Here, single-edge failure refers to graph failures with only one failed edge at a time [5]. Note that, other types of edge failure, such as dual-failure in [13], may allow multiple failed edges at a time. But they are considered much harder than single-failure [13]. To shed light on these challenging issues, we focus on single-edge failures in this paper.

Contributions. Since 2-hop labeling has shown its power to support instant responses to shortest path distance queries on stable graphs, our work aims at extending this technique to support unstable graphs. Existing shortest path indexing techniques based on 2-hop labeling can be used to precompute the whole shortest path index for a graph. The resulted indexes can normally answer distance queries fast using moderate storage space [2, 15]. However, applying indexing techniques designed for static/stable graphs directly to evolving/unstable graphs may lead to inefficiency. When considering every single-edge failure case and constructing a corresponding index for each case, the size of all these indexes will become too big to manage. For instance, a snapshot of the Gnutella peer-to-peer (P2P) file sharing network in August 2002 contains more than 6,000 vertices[1] and 20,000 edges. Using state-of-the-art method, Pruned Landmark Labeling (PLL) [2], the index size is slightly more than 5 MB. However, suppose we want to construct such index for each single-edge failure case, the total index size would be more than $5 \times 20,000 = 10^5$ MB.

To address the deficiency of existing shortest path indexing techniques, this paper proposes a generic framework named SIEF, a Supplemental Index for Edge Failures on a graph, to construct compact shortest path indexes efficiently for unstable graphs where single-edge failures may exist. As an initial attempt on this challenging issue, we focus on unweighted, undirected graphs. Similar to other distance labeling based indexing methods [2, 15], our method can be extended to weighted and/or directed graphs. We highlight our main contributions in the following.

- We present the concept of *well-ordering 2-hop distance labeling* and identify its important properties that can be utilized to design algorithms for shortest path indexes on graphs with edge failures.

- We analyze shortest path index constructions on graphs with edge failures theoretically. We develop the corresponding theorems as well as novel algorithms to enable constructions of compact indexes for all the single-edge failure cases of the entire graph. By applying our approach to the aforementioned Gnutella P2P dataset, the size of the generated SIEF index together with the original index created for the original graph is merely 14 MB, which is much more compact than 10^5 MB by directly using PLL method [2] to construct indexes for each single-edge failure case.

- We conduct extensive experiments on six real-world graphs to verify the efficiency and effectiveness of our method. The results show that our method can efficiently answer shortest path distance queries avoiding a failed edge with very compact labeling indexes.

The rest of this paper is organized as follows. In Section 2, we review the related work. In Section 3, we present some preliminaries on 2-hop distance labeling. We then present the framework and the details of our approach in Section 4. In Section 5, we report the results of an extensive experimental study using six graphs from real-world. Finally, we present some concluding remarks in Section 6.

2. RELATED WORK

In this section, we review the major techniques that are most closely related to our work.

Distance labeling has been an active research area in recent years. In [10], Cheng and Yu exploit the strongly connected components property and graph partitioning to precompute 2-hop distance cover. However, the graph partitioning process introduces high cost because it has to find vertex separators recursively. Hierarchical hub labeling (HHL) proposed by Abraham et al. [1] is based on the partial order of vertices. Smaller labeling results can be obtained by computing labeling for different partial order of vertices. In [18], Jin et al. propose a highway-centric labeling (HCL) that uses a spanning tree as a highway. Based on the highway, a 2-hop labeling is generated for fast distance computation.

Very recently, the pruned landmark labeling (PLL) [2] is proposed by Akiba et al. to pre-compute 2-hop distance labels for vertices by performing a breadth-first search from every vertex. The key idea is to prune vertices that have obtained correct distance information during breadth-first searches, which helps reduce the search space and sizes of labels. Further, query performance is also improved as the number of label entries per vertex is reduced. IS-Label (or ISL) is developed by Fu et al. in [15] to pre-compute 2-hop

[1] http://snap.stanford.edu/

distance label for large graphs in memory constrained environments. ISL is based on the idea of independent set of vertices in a large graph. By recursively removing an independent set of vertices from the original graph, and by augmenting edges that preserve distance information after the removal of vertices in the independent set, the remaining graph keeps the distance information for all remaining vertices in the graph. Apart from the 2-hop distance labeling technique, a multi-hop distance labeling approach [8] is also studied, which can reduce the overall size of labels at the cost of increased distance querying time.

Tree decomposition approach has been recently investigated [23, 4] for answering distance queries on graphs. Wei proposes TEDI [23], which first decomposes a graph into a tree and then constructs a tree decomposition for the graph. A tree decomposition of a graph is a tree with each vertex associated with a set of vertices in the graph, which is also called a *bag*. The shortest paths among vertices in the same bag are pre-computed and stored in bags. For any given source and target vertices, a bottom-up operation along the tree can be executed to find the shortest path. An improved TEDI index is proposed by Akiba et al. in [4] that exploits a core-fringe structure to improve index performance. However, due to the large size of some bags in the decomposed tree, the construction time for a large graph is costly and thus such indexing approaches cannot scale well.

Maintenance of 2-hop reachability labeling is also studied. For example, HOPI (2-HOP-cover-based Index) introduces some maintenance techniques for the constructed index. HOPI is developed by Schenkel et al. in [20] and is designed to speed up connection or reachability tests in XML documents based on the idea of 2-hop cover. HOPI is able to update indexes for insertions and deletions of nodes, edges or even XML documents. To the best of our knowledge, HOPI is the first work on maintenance of 2-hop labeling. Recently, maintenance of 2-hop labeling for large graphs has also been studied by Bramandia et al. in [7]. However, all these studies focus on reachability queries and are based on 2-hop labeling but not on 2-hop distance labeling.

Incremental maintenance of 2-hop distance labeling is also studied very recently by Akiba et al. in [3]. In that work, incremental updates (i.e., edge insertions) of 2-hop labeling indexes are investigated. To support fast incremental updates, outdated distance labels are kept, which will not affect the distance computation on the updated graphs in the incremental case. However, for the decremental case (i.e., edge deletions), this approach will not work, as outdated distance labels must be removed first and then some necessary labels of the 2-hop labeling index need to be recomputed. Hence, their update algorithms cannot be applied on edge deletions (i.e., edge failures), which will be discussed in this paper.

3. PRELIMINARIES

3.1 2-Hop Distance Labeling

The technique of 2-hop cover can be used to solve reachability problems (using reachability labels) and shortest path distance querying problems (using distance labels) on graphs [12]. Since our work focuses on the shortest path distance querying problems, we adopt distance labels with the 2-hop cover technique. We specifically refer to it as *2-hop distance labeling* or *2-hop distance cover*.

Assume a graph $G = (V, E)$, where V is a set of vertices and E is a set of edges. For each vertex $v \in V$, there is a pre-computed label $L(v)$, which is a set of vertex and distance pairs (u, δ_{uv}). Here u is a vertex and δ_{uv} is the shortest path distance between u and v. Given such a labeling for all vertices in G, denoted by L, for any pair of vertices s and t in G, we have

$$\begin{aligned} dist(s,t,L) =& min\{\delta_{vs} + \delta_{vt} | (v, \delta_{vs}) \in L(s) \\ & \text{and } (v, \delta_{vt}) \in L(t)\} \end{aligned} \quad (1)$$

If $L(s)$ and $L(t)$ do not share any vertices, we have $dist(s,t,L)$ $=\infty$. The distance between any given vertices s and t in G is denoted by $d_G(s,t)$. If we have $d_G(s,t) = dist(s,t,L)$ for all s and t in G, we call the labeling result L a 2-hop distance cover.

3.2 Well-Ordering 2-Hop Distance Labeling

For a connected graph G, there exists a sequence of vertices $\sigma =< v_0, v_1, v_2, \ldots, v_{n-1} >$. We denote the order of any vertex v_i as $\sigma[v_i]$ and we have $\sigma[v_i] = i$ for the above given vertex sequence. Based on this, we can define *Well-Ordering 2-Hop Distance Labeling* in the following.

Definition 1 (***Well-Ordering 2-Hop Distance Labeling***). Suppose that (1) each vertex v_i has a distance labeling $L(v_i)$, and the labeling result L of all vertices forms a 2-hop distance cover of G; (2) for any pair of vertices v_i and v_j, given that $\sigma[v_i] < \sigma[v_j]$, then v_j is not in $L(v_i)$ and v_i may be in $L(v_j)$. We call such a 2-hop distance cover a *well-ordering 2-hop distance labeling.* Alternatively we say that a 2-hop distance cover has well-ordering property.

Similar concepts of well-ordering 2-hop distance labeling also appear in recent research efforts such as HHL [1], PLL [2], and ISL [15]. This confirms that well-ordering 2-hop distance labeling is important in the related research area. More importantly, we will show in this paper that the well-ordering property is also a basic concept in the design of index construction algorithms for distance labeling computation on unstable graphs where edges may fail.

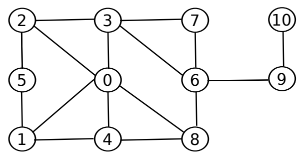


Figure 1: A graph example

In a graph containing multiple connected components, suppose its 2-hop labeling is L. For any pair of vertices u and v in different connected components, we can assert that $L(u)$ and $L(v)$ do not share any vertex according to the definition of 2-hop cover. Each connected component has its own vertex orders. For such a graph, we will have separate vertex orders for each connected component. We denote a connected component containing vertex u as $C(u)$. If u and v belong to the same connected component, we have $C(u) = C(v)$.

Figure 1 shows an example graph with 11 vertices and Table 1 shows a well-ordering 2-hop distance labeling result L for the graph (L can be constructed by PLL [2] using the same vertex ordering as that specified in the table). In the

Table 1: 2-Hop Distance Labeling L for Figure 1

Label	Entries
$L(0)$	(0,0)
$L(1)$	(0,1) (1,0)
$L(2)$	(0,1) (2,0)
$L(3)$	(0,1) (2,1) (3,0)
$L(4)$	(0,1) (1,1) (4,0)
$L(5)$	(0,2) (1,1) (2,1) (5,0)
$L(6)$	(0,2) (2,2) (3,1) (4,2) (6,0)
$L(7)$	(0,2) (2,2) (3,1) (6,1) (7,0)
$L(8)$	(0,1) (4,1) (6,1) (8,0)
$L(9)$	(0,3) (2,3) (3,2) (4,3) (6,1) (9,0)
$L(10)$	(0,4) (2,4) (3,3) (4,4) (6,2) (9,1) (10,0)

table, the order of vertices is $< 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 >$. Take $L(5)$ as an example to further explain the idea of well-ordering 2-hop distance labeling. $L(5)$ is the label of vertex 5. By the well-ordering property, label entries in $L(5)$ can only contain vertices $0, 1, 2, 3, 4$ and 5. Since label entries containing vertices 3 and 4 are redundant in $L(5)$ (this will be explained in more details later in this section), label entries in $L(5)$ only contain vertices $0, 1, 2$ and 5.

3.3 Properties of Well-Ordering 2-Hop Distance Labeling

Technically speaking, if we index shortest paths for all pairs using a labeling method, we will obtain an index that occupies $O(n^2)$ disk space. This index can be considered as a special 2-hop distance labeling. Obviously, the space complexity of this is too high for large graphs. Constructing a minimal 2-hop distance labeling has been proven to be NP-hard [12]. Therefore, an alternative way to obtain labeling results with reduced sizes is by using heuristic methods [10, 8, 2, 15]. Well-ordering 2-hop distance labeling is one of the techniques that can help to design efficient algorithms for constructing shortest path distance labeling indexes and for index maintenance. We identify its useful properties in the following.

LEMMA 1. *Given a well-ordering 2-hop distance labeling L of a connected graph G, suppose $u \in G$ and $\sigma[u]$ is a minimum among all vertices in G, then for any vertex $v \in G$, we must have $(u, \delta_{uv}) \in L(v)$.*

PROOF. It is trivial to prove this when $v = u$ since $(u, 0) \in L(u)$. We prove the case when $v \neq u$ by contradiction. Suppose there exists a vertex $v \in G$, $(u, \delta_{uv}) \notin L(v)$. By the definition of L, since $\sigma[u]$ is minimum, $L(u)$ will contain only one label entry $(u, 0)$. Then it is obvious that $L(u)$ and $L(v)$ do not share any vertex, which leads to $dist(u, v, L) = \infty$. This implies that u and v belong to different connected components, which is false. Therefore, the lemma is proved. □

LEMMA 2. *Given a well-ordering 2-hop distance labeling L of a connected graph G, suppose $s, t, u \in G$ and $dist(s, t, L) = dist(s, u, L) + dist(u, t, L)$, then u must be an internal vertex of a certain shortest path between s and t.*

PROOF. Since $dist(s, t, L) = dist(s, u, L) + dist(u, t, L)$, there must exist some shortest path that starts from s, passes u, and ends at t. Hence the lemma is proved. □

Take vertices 5, 6 and 2 in Figure 1 as an example. From Table 1, we have $dist(5, 6, L)$=3 and $dist(5, 2, L)+dist(2, 6, L)$ =1+2=3. From Figure 1, we can see that vertex 2 is an internal vertex on some shortest path, denoted as p, between vertex 5 and vertex 6. In this case, we have $p =< 5, 2, 3, 6 >$.

LEMMA 3. *Given a well-ordering 2-hop distance labeling L of a connected graph G, suppose $s, t, u \in G$ and u has minimum vertex order $\sigma[u]$ among all shortest paths between s and t. Then we must have $(u, \delta_{us}) \in L(s)$ and $(u, \delta_{ut}) \in L(t)$ and $dist(s, t, L) = \delta_{us} + \delta_{ut}$.*

PROOF. We prove this by contradiction. Without loss of generality, suppose $(u, \delta_{us}) \notin L(s)$. In order to calculate $dist(s, u, L)$, there must exist some vertex v other than u, where $(v, \delta_{vs}) \in L(s)$, $(v, \delta_{vu}) \in L(u)$ and $dist(s, u, L) = \delta_{vs} + \delta_{vu}$. According to Lemma 2, v must be an internal vertex of some shortest path between s and u. Hence v must also be an internal vertex of some shortest path between s and t. Meanwhile, by definition, we must have $\sigma[v] < \sigma[u]$. This contradicts our assumption that u has the minimum vertex order among all shortest paths between s and t. Hence, we must have $(u, \delta_{us}) \in L(s)$. Furthermore, u is an internal vertex of some shortest path between s and t, thus $dist(s, t, L) = \delta_{us}+\delta_{ut}$. Hence the lemma is proved. □

Take vertices 1 and 6 in Figure 1 as an example. Paths $p_1 =< 1, 0, 8, 6 >$, $p_2 =< 1, 0, 3, 6 >$ and $p_3 =< 1, 4, 8, 6 >$ are all the shortest paths between vertices 1 and 6. Vertex 0 is the one with minimum order along all these paths. From Table 1 we can see that both vertices 1 and 6 contain a label entry $(0, \delta)$. We can also easily check that $dist(1, 6, L) = \delta_{0,1} + \delta_{0,6} = 1 + 2 = 3$.

LEMMA 4. *Given a well-ordering 2-hop distance labeling L of a connected graph G, suppose $\sigma[u] < \sigma[v]$. If there is a label entry $(u, \delta_{uv}) \in L(v)$, we must have for any label entry $(r, \delta_{rv}) \in L(v)$, (1) $\delta_{uv} \leq \delta_{rv} + dist(r, u, L)$; (2) if $\sigma[r] < \sigma[u]$ and $\delta_{uv} = \delta_{rv} + dist(r, u, L)$ then $(u, \delta_{uv}) \in L(v)$ is a redundant label entry.*

PROOF. We first prove the first claim that $\delta_{uv} \leq \delta_{rv} + dist(r, u, L)$. By definition and the triangle inequalities we must have $\delta_{uv} = d_G(u, v) = dist(u, v, L) \leq \delta_{rv}+dist(r, u, L)$.

We then prove the second claim. We need to prove that if $\delta_{uv} = \delta_{rv} + dist(r, u, L)$, then for any vertex t, when we calculate $dist(v, t, L)$, (u, δ_{uv}) in $L(v)$ is not required. For t, there are three cases: (1) $(u, \delta_{ut}) \notin L(t)$; (2) $(u, \delta_{ut}) \in L(t)$ but $\delta_{uv} + \delta_{ut} > dist(v, t, L)$; (3) $(u, \delta_{ut}) \in L(t)$ and $\delta_{uv} + \delta_{ut} = dist(v, t, L)$. For Case (1) and Case (2), it is trivial since (u, δ_{uv}) in $L(v)$ is not required to calculate $dist(v, t, L)$. For Case (3), according to Lemma 2, u is an internal vertex of some shortest paths between v and t. Similarly, since $\delta_{uv} = \delta_{rv} + dist(r, u, L)$, r is an internal vertex of some shortest paths between u and v, which means r is also an internal vertex of some shortest paths between v and t. In such case, we prove in the following that there must exist a vertex s other than u and we have $(s, \delta_{sv}) \in L(v)$, $(s, \delta_{st}) \in L(t)$ where $\delta_{st} + \delta_{sv} = dist(v, t, L)$.

Suppose s is the vertex with minimum vertex order among all shortest paths between v and t. According to Lemma 3, we must have $(s, \delta_{sv}) \in L(v)$, $(s, \delta_{st}) \in L(t)$ and $\delta_{st} + \delta_{sv} = dist(v, t, L)$. Since $\sigma[s] \leq \sigma[r] < \sigma[u]$, s is not the same vertex of u. Therefore, (u, δ_{uv}) in $L(v)$ is not required to calculate $dist(v, t, L)$. Hence, the second claim is also proved. □

Take label entries of vertex 5 in Table 1 as an example. We have $\sigma(3) < \sigma(5)$ and $\sigma(2) < \sigma(3)$. We also have $\delta_{3,5} = 2 = \delta_{2,5} + \delta_{2,3}$. Therefore $(3, 2)$ is a redundant label entry in $L(5)$, which can be removed from $L(5)$.

4. THE SIEF APPROACH

In this section, we first provide an overview of our SIEF approach. We then analyze the 2-hop distance labeling computation on graphs with single-edge failures and introduce a set of algorithms to achieve fast and compact index constructions.

4.1 SIEF Overview

After an edge fails on a graph, we observe that distances of a considerable proportion of shortest paths between any pair of vertices remain unchanged. Therefore, to construct a new index for each single-edge failure case, we only need to compute new labels for those vertices with changed shortest path distances due to the edge failure. Overall, our index construction approach can be divided into two main stages. In the first stage, *IDENTIFY*, we identify affected vertices after an edge fails. In the second stage, *RELABEL*, we relabel all affected vertices with necessary additional label entries for the single-edge failed graph. These new label entries form a new part of the index, which is called a *supplemental index*.

Before the detailed discussions of our algorithms, suppose that the failed edge is (u, v) in G, and the new graph is G', we introduce a concept for the supplemental index construction:

Definition 2 (***Affected vertices*** $AV_{(u,v)}$). For any vertices s and t, if $d_{G'}(s, t) \neq d_G(s, t)$, then $s \in AV_{(u,v)}$ and $t \in AV_{(u,v)}$.

To be specific, $AV_{(u,v)}$ contains all vertices whose distance to some other vertex must have been changed due to the failed edge (u, v). It is quite clear that supplemental indexes should be constructed to maintain all new distances for each single-edge failure case. In other words, supplemental indexes are constructed based on all the vertices in $AV_{(u,v)}$. Further, in order to be compact, the supplemental indexes should only answer distances that cannot be answered by the original index.

4.2 Identification of Affected Vertices

Before we can start to construct supplemental indexes, we need to identify all the affected vertices in $AV_{(u,v)}$ first. A naive method would be to compare distances for any possible pair of affected vertices in the original graph G and the new graph G' with a failed edge (u, v), but that would be very time consuming as it will need to test distances of $O(n^2)$ pairs of vertices. In the following, we will try to identify some important properties for vertices in $AV_{(u,v)}$ for us to identify $AV_{(u,v)}$ more efficiently and accurately.

LEMMA 5. *After removing the failed edge (u, v) from graph G, for any vertex s, t in G', we must have $d_{G'}(s, t) \geq dist(s, t, L)$.*

PROOF. In the old graph G, there are only two types of shortest paths: (1) shortest paths containing edge (u, v); and (2) shortest paths not containing edge (u, v). For the former, we have $d_{G'}(s, t) \geq d_G(s, t) = dist(s, t, L)$. For the latter, we have $d_{G'}(s, t) = d_G(s, t) = dist(s, t, L)$. Thus the lemma is proved. □

LEMMA 6. *After removing the failed edge (u, v) from graph G, for any vertex s, t in G', if $d_{G'}(s, t) > dist(s, t, L)$, and suppose a shortest path between s and t in G is $\pi_G(s, t)$, then we must have $uv \in \pi_G(s, t)$ or $vu \in \pi_G(s, t)$.*

PROOF. This can be proved by contradiction. Suppose we have $d_{G'}(s, t) > dist(s, t, L)$ but $uv \notin \pi_G(s, t)$ and $vu \notin \pi_G(s, t)$, which means edge (u, v) does not appear in $\pi_G(s, t)$. In such case, there must exist a path $P_{G'}(s, t)$ in G' where $\pi_G(s, t) = P_{G'}(s, t)$. This means $d_{G'}(s, t)$ must be at most the length of $P_{G'}(s, t)$, i.e., the length of $\pi_G(s, t)$. Thus, we must have $d_{G'}(s, t) = dist(s, t, L') \leq d_G(s, t)$. This contradicts our assumption $d_{G'}(s, t) > dist(s, t, L)$. □

According to Lemma 6 and the definition of affected vertices, if we have $d_{G'}(s, t) > dist(s, t, L) = d_G(s, t)$, we must have that $s, t \in AV_{(u,v)}$. This further means the shortest path(s) between s and t in the original graph G must contain the failed edge (u, v). Then, after edge (u, v) fails, take any one of these shortest paths (if multiple shortest paths exist; if not, we will have one and only one shortest path containing (u, v)) as an example, denoted as $\pi_G(s, t)$. Then it is easy to imagine that $\pi_G(s, t)$ will become two segments: one segment ends at u, denoted as Seg_u and the other segment ends at v, denoted as Seg_v. Without loss of generality, suppose s falls on the Seg_u and t falls on Seg_v. Since Seg_u and Seg_v must also be shortest paths from s to u and from t to v, respectively, this means we must have $d_G(s, u) = d'_G(s, u)$ and $d_G(t, v) = d'_G(t, v)$. But in the meantime, we must have $d_G(s, v) \neq d'_G(s, v)$ and $d_G(t, u) \neq d'_G(t, u)$ since otherwise we will have $d_{G'}(s, t) = d_G(s, t)$, which is impossible. Based on this observation, we can see that vertices in $AV_{(u,v)}$ form two disjoint sets: one set is $AV_{(u,v)}(u)$ and the other set is $AV_{(u,v)}(v)$, where for $\forall s \in AV_{(u,v)}(u)$ and $\forall t \in AV_{(u,v)}(v)$, we must have $d_{G'}(s, t) > d_G(s, t)$, $d_G(s, u) = d'_G(s, u)$ and $d_G(t, v) = d'_G(t, v)$. Since (u, v) is the failed edge, obviously, we must have $u \in AV_{(u,v)}(u)$ or $v \in AV_{(u,v)}(v)$. Further, it should be noted that, $\forall s, t \in AV_{(u,v)}(u)$, we must have $d_G(s, t) = d'_G(s, t)$. The same conclusion can be made on $\forall s, t \in AV_{(u,v)}(v)$.

Next, we are going to show that all vertices $s \in AV_{(u,v)}(u)$ form a tree rooted at u and similarly, all vertices $t \in AV_{(u,v)}(v)$ also form a tree rooted at v.

LEMMA 7. *After removing the failed edge (u, v), for any vertex w in G', suppose w is an affected vertex, i.e. $w \in AV_{(u,v)}(u)$ or $w \in AV_{(u,v)}(v)$. Without loss of generality, we assume $w \in AV_{(u,v)}(u)$. Then we must have $d_G(w, v) = d_G(w, u) + 1$.*

PROOF. Since $w \in AV_{(u,v)}(u)$, we must have that $d_G(w, v) \neq d'_G(w, v)$, which means that any shortest path between w and v in G, denoted as p_{wv} must contain the failed edge (u, v), which must also be a shortest path between w and v. Hence, there must exist a certain shortest path between w and v containing edge (u, v) in the original graph and we can denote it as $p_{wv} = p_{wu} + (u, v)$. Hence, we must have $d_G(w, v) = d_G(w, u) + 1$. □

LEMMA 8. *After removing the failed edge (u, v), suppose w in G' is an affected vertex, i.e. $w \in AV_{(u,v)}(u)$ or $w \in AV_{(u,v)}(v)$. Without loss of generality, we assume $w \in AV_{(u,v)}(u)$. Then there must exist a certain shortest path between w and v containing edge (u, v) in the original graph, where each internal vertex is an affected vertex in $AV_{(u,v)}(u)$.*

Algorithm 1 Identify affected vertices

```
Input: $G$, $(u, v)$, distance vectors $d_u, d_v, d'_u, d'_v$
Output: $AV_{(u,v)}(u)$, $AV_{(u,v)}(v)$
 1: Initialize flag $m[t] \leftarrow 0$ for any vertex $t$ in $G$
 2: $m[u] \leftarrow 1$
 3: $Q \leftarrow \emptyset$
 4: Enqueue $u$ into $Q$
 5: while $Q$ is not empty do
 6:     Dequeue $t$ from $Q$
 7:     for all neighbor vertex $r$ of $t$ do
 8:         if $m[r] = 0$ then
 9:             if $d_v[r] = d_u[r] + 1$ and $d'_v[r] \neq d_u[r] + 1$ then
10:                 $AV_{(u,v)}(u) \leftarrow AV_{(u,v)}(u) \cup \{r\}$
11:                 Enqueue $r$ into $Q$
12:             $m[r] \leftarrow 1$
13: Repeat the above steps by mapping $u \leftarrow v$ and $v \leftarrow u$
    to identify $AV_{(u,v)}(v)$
```

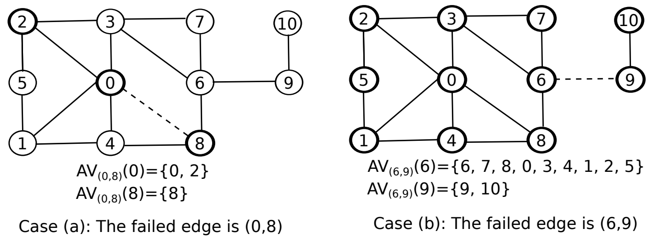

AV(0,8)(0)={0, 2}
AV(0,8)(8)={8}
Case (a): The failed edge is (0,8)
AV(6,9)(6)={6, 7, 8, 0, 3, 4, 1, 2, 5}
AV(6,9)(9)={9, 10}
Case (b): The failed edge is (6,9)

Figure 2: Affected vertices identification

PROOF. Since $w \in AV_{(u,v)}(u)$, then according to Lemma 7, we must have the fact that any shortest path between w and u, denoted as p_{wu}, plus edge (u, v) in the original graph must be a shortest path between w and v. Then, there must exist a certain shortest path between w and v containing edge (u, v) in the original graph and we can denote it as $p_{wv} = p_{wu} + (u, v)$.

It is clear that the internal vertices of p_{wv} must also be on some shortest path p_{wu}. And all shortest paths from these internal vertices to vertex v must contain edge (u, v), which means, their distances to vertex v must have changed in the new graph G'. Therefore, they must also be affected vertices in $AV_{(u,v)}(u)$ like w. □

Note that, according to Lemma 8, $AV_{(u,v)}(u)$ and $AV_{(u,v)}(v)$ can be considered as trees rooted at u and v, respectively. Moreover, we must have $AV_{(u,v)}(u) \bigcap AV_{(u,v)}(v) = \emptyset$. This is because otherwise, any vertex r in $AV_{(u,v)}(u) \bigcap AV_{(u,v)}(v)$ must have $d_G(r, v) = d_G(r, u)+1$ and $d_G(r, u) = d_G(r, v)+1$, which is impossible. Lemma 8 forms the basis of Algorithm 1. Note that, in Algorithm 1, we need to calculate distance vectors d_u, d_v, d'_u and d'_v for each single-edge failure case. Here, d_u stores distances from all vertices in G to vertex u while d'_u stores distances from all vertices in G' to vertex u. Distance vectors d_v and d'_v are similar. The calculations can be done efficiently using a BFS algorithm. To reduce the calculation cost, we will fix an end point of failed edges, i.e., we will firstly compute affected vertices for all edges attached to u then we move to other vertices for processing the rest single-edge failure cases.

Figure 2 shows two examples of identifying affected vertices. It uses the same graph in Figure 1. In this figure, the first example is Case (a), where the failed edge is $(0, 8)$. The second example is Case (b), where the failed edge is $(6, 9)$. In Case (a), starting from vertex 0, we identify the affected vertex set rooted at 0 as $AV_{(0,8)}(0) = \{0, 2\}$ since only vertices 0 and 2 have changed their distance to vertex 8. Meanwhile, starting from vertex 8, we identify the affected vertex set rooted at 8 as $AV_{(0,8)}(8) = \{8\}$ since only vertex 8 has changed its distance to vertex 0. Differently, in Case (b), as can be observed in the figure, the original graph will become two connected components rooted at vertices 6 and 9, respectively. In this case, it is obvious that we have $AV_{(6,9)}(6) = \{6, 7, 8, 0, 3, 4, 2, 5\}$ and $AV_{(6,9)}(9) = \{9, 10\}$.

4.3 Relabeling: Supplemental Index Construction

After identifying all affected vertices, we can start relabeling the affected vertices in order for fast computation of shortest path distances on the graph with single-edge failures. Only supplemental indexes will be created, i.e., only changed distance information will be captured in supplemental indexes. All the unchanged distance information will be still computed using the original indexes (such as the distance labeling in Table 1 for the example graph in Figure 1). We develop two relabeling algorithms for the supplemental index construction, namely the BFS AFF algorithm and the BFS ALL algorithm. Detailed descriptions of these two algorithms are presented in the following.

4.3.1 BFS AFF algorithm

The BFS AFF algorithm relabels affected vertices using the traditional BFS algorithm. The BFS AFF algorithm uses a *late label-pruning* strategy which can save memory usage during the relabeling process. The detail steps are shown in Algorithm 2. To help understand the main idea of the BFS AFF algorithm, Figure 3 also depicts an example of the supplemental index construction process using the BFS AFF algorithm.

The failed edge is (0,8) in this example and there are three steps in Figure 3. Each step relabels one affected vertex. At Step (1), BFS AFF algorithm performs BFS from vertex 0. The number beside each node is the distance from that node to the BFS root, vertex 0. In this step, vertex 8 is the only affected vertex in $AV_{(0,8)}(8)$ that has larger vertex order than vertex 0. Therefore, the BFS process starting from vertex 0 will stop at distance 2 and will not examine vertices 9 and 10. After the BFS process stops, we add a supplemental label entry to the supplemental label of vertex 8, resulting in $SL_{(0,8)}(8) = \{(0, 2)\}$. At Step (2), BFS process starts from vertex 2. Note that, the distance information has been discarded at this step. Similarly, vertex 8 is the only affected vertex in $AV_{(0,8)}(8)$ that has larger vertex order than vertex 2. Then the BFS process starting from vertex 2 will stop at distance 3. Then we may want to add another label entry $(2, 3)$ into $SL_{(0,8)}(8)$. But based on the original index shown in Table 1 and the current supplemental label $SL_{(0,8)}(8) = \{(0, 2)\}$, we find that $(2, 3)$ is a redundant label entry in $SL_{(0,8)}(8) = \{(0, 2)\}$ since the distance between vertex 2 and vertex 8 can be computed based on $SL_{(0,8)}(8) = \{(0, 2)\}$ and the original index in Table 1. We call this the late-pruning strategy. Finally, at Step (3), the BFS process will start from vertex 8. However, since no vertex in $AV_{(0,8)}(0)$ has smaller vertex order than vertex 8, no label entry will be added to the supplemental index at this step. The final supplemental index that is constructed

Algorithm 2 BFS AFF algorithm

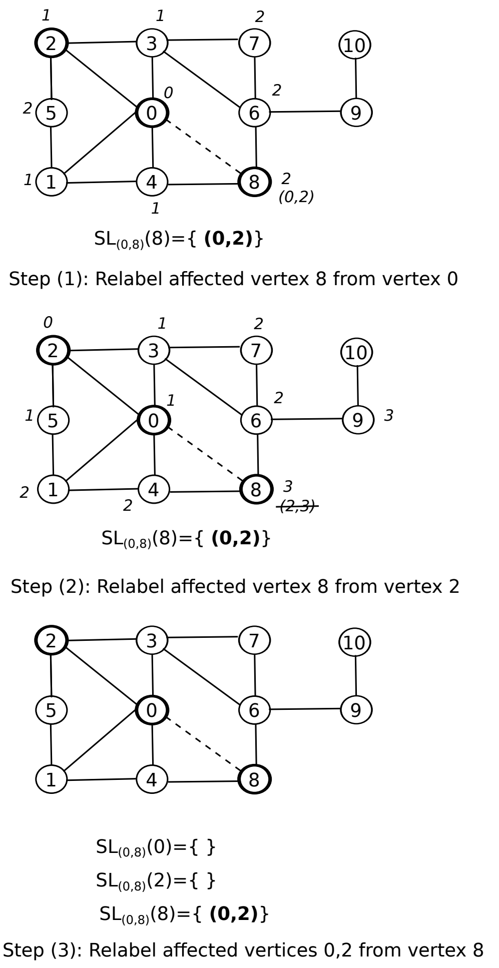
```
Input: $G$, $(u,v)$, $AV_{(u,v)}(u)$, $AV_{(u,v)}(v)$
Output: The supplemental index $SI_u$ and $SI_v$ for the edge
    failure case of $(u,v)$
 1: $G' \leftarrow G - \{(u,v)\}$
    //Construct $SI_u$ for vertices in $AV_{(u,v)}(u)$
 2: $SI_u \leftarrow \emptyset$
 3: for all $r \in AV_{(u,v)}(u)$ (in ascending vertex order) do
 4:     Initialize supplemental label for $r$: $SL \leftarrow \emptyset$
 5:     Start BFS algorithm to compute all the distances from
        $r$ to any vertices in $AV_{(u,v)}(v)$ that have larger vertex
        order than $\sigma(r)$
 6:     for all vertex t in $AV_{(u,v)}(v)$ that has $\sigma(t) > \sigma(r)$ do
 7:         if $(t, d_{G'}(t,r))$ is not a redundant label entry in $SL$
            then
 8:             $SL \leftarrow SL \cup (t, d_{G'}(t,r))$
 9:     $SI_u \leftarrow SI_u \cup (r, SL)$
    //Construct $SI_v$ for vertices in $AV_{(u,v)}(v)$
10: $SI_v \leftarrow \emptyset$
11: for all $r \in AV_{(u,v)}(v)$ (in ascending vertex order) do
12:     Initialize supplemental label for $r$: $SL \leftarrow \emptyset$
13:     Start BFS algorithm to compute all the distances from
        $r$ to any vertices in $AV_{(u,v)}(u)$ that have larger vertex
        order than $\sigma(r)$
14:     for all vertex t in $AV_{(u,v)}(u)$ that has $\sigma(t) > \sigma(r)$
        do
15:         if $(t, d_{G'}(t,r))$ is not a redundant label entry in $SL$
            then
16:             $SL \leftarrow SL \cup (t, d_{G'}(t,r))$
17:     $SI_v \leftarrow SI_v \cup (r, SL)$
```

for the failed edge $(0,8)$ on the graph shown in Figure 1 is shown at Step (3). We will show later in Section 4.4 that such supplemental index is adequate for distance query evaluation.

4.3.2 BFS ALL algorithm

The BFS ALL algorithm is very similar to the BFS AFF algorithm. The major diference is that the BFS ALL algorithm uses an *early label-pruning* strategy which consumes more memory during the relabeling process but gains acceleration of the relabeling process. The detail steps are shown in Algorithm 3. Figure 4 also depicts an example of the supplemental index construction process using the BFS ALL algorithm.

The failed edge is also (0,8) in this example and there are three steps in Figure 4. The main difference between BFS ALL an BFS AFF algorithms is that, in the BFS ALL algorithm, the distance information will be kept at each BFS step, using a set of temporary labels stored in TL. This distance information in TL can be used to prune label entries at the later BFS steps of the index construction process for all vertices in the graph and some vertices can be pruned during a BFS process. For example, at Step (2) in Figure 4, the number of vertices we need to visit (the vertices with bold label entries) is only seven, while at Step (2) in Figure 3, that number is 10 (by counting the vertices with distance information). Therefore, three vertices can be pruned at Step (2) in the BFS ALL algorithm. We call this the early-pruning strategy. It is obvious that the BFS ALL algorithm introduces more memory usage since BFS ALL has TL while BFS AFF does not. But the benefit of TL is that it



$SL_{(0,8)}(8)$={ **(0,2)**}
Step (1): Relabel affected vertex 8 from vertex 0
(0,2)
$SL_{(0,8)}(8)$={ **(0,2)**}
Step (2): Relabel affected vertex 8 from vertex 2
~~(2,3)~~
$SL_{(0,8)}(0)$={ }
$SL_{(0,8)}(2)$={ }
$SL_{(0,8)}(8)$={ **(0,2)**}
Step (3): Relabel affected vertices 0,2 from vertex 8

Figure 3: Supplemental index construction: BFS AFF on failed edge $(0,8)$

can prune vertices at an early stage, and as will be shown in Section 5, this can speed up the BFS process greatly. Nevertheless, the final supplemental index constructed by the BFS ALL algorithm is the same as that constructed by the BFS AFF algorithm as the construction of SI_u and SI_v in both algorithms is the same.

4.4 Distance Query Evaluation on SIEF

For each single-edge failure case, we classify all possible distance queries into different types. Suppose the graph is G, the original labeling index is L, the failed edge is (u,v), the affected vertices are in $AV_{(u,v)}(u)$ and $AV_{(u,v)}(v)$, and the supplemental index is $SI_{(u,v)}$ (here, $SI_{(u,v)} = SI_u \cup SI_v$). We also denote $G' = G - \{(u,v)\}$. Given any pair of vertices s,t, we would like to compute the distance between s,t on G', denoted as $d_{G'}(s,t)$. Then we have the following different cases:

- Case 1: $s \notin AV_{(u,v)}(u) \cup AV_{(u,v)}(v)$ and $t \notin AV_{(u,v)}(u) \cup AV_{(u,v)}(v)$
- Case 2: $s \notin AV_{(u,v)}(u) \cup AV_{(u,v)}(v)$ and $t \in AV_{(u,v)}(u) \cup AV_{(u,v)}(v)$, or similarly, $s \in AV_{(u,v)}(u) \cup AV_{(u,v)}(v)$ and $t \notin AV_{(u,v)}(u) \cup AV_{(u,v)}(v)$

Algorithm 3 BFS ALL algorithm

```
Input: $G$, $(u,v)$, $AV_{(u,v)}(u)$, $AV_{(u,v)}(v)$
Output: The supplemental index $SI_u$ and $SI_v$ for the edge
    failure case of $(u,v)$
 1: $G' \leftarrow G - \{(u,v)\}$
    //Construct $SI_u$ for vertices in $AV_{(u,v)}(u)$
 2: $SI_u \leftarrow \emptyset$
 3: Initialize temporary labels $TL \leftarrow \emptyset$
 4: for all $r \in AV_{(u,v)}(u)$ (in ascending vertex order) do
 5:     Initialize supplemental label for $r$: $SL \leftarrow \emptyset$
 6:     Start BFS algorithm to compute all the distances from
        $r$ to any vertices in $AV_{(u,v)}(v)$ that have larger vertex
        order than $\sigma(r)$ and record all temporary labels for all
        encountered vertices in $TL$; during the BFS process,
        if a new temporary label entry for a vertex $w$ is redundant
        in $TL$, all neighbor vertices of $w$ can be ignored
        by BFS
 7:     for all vertex t in $AV_{(u,v)}(v)$ that has $\sigma(t) > \sigma(r)$ and
        has been searched by the above BFS process do
 8:         if $(t, d_{G'}(t,r))$ is not a redundant label entry in $SL$
            then
 9:             $SL \leftarrow SL \cup (t, d_{G'}(t,r))$
10:     $SI_u \leftarrow SI_u \cup (r, SL)$
    //Construct $SI_v$ for vertices in $AV_{(u,v)}(v)$
11: $SI_v \leftarrow \emptyset$
12: Initialize temporary labels $TL \leftarrow \emptyset$
13: for all $r \in AV_{(u,v)}(v)$ (in ascending vertex order) do
14:     Start BFS algorithm to compute all the distances from
        $r$ to any vertices in $AV_{(u,v)}(u)$ that have larger vertex
        order than $\sigma(r)$ and record all temporary labels for
        all encountered vertices in $TL$; during the BFS process,
        if a new temporary label entry for a vertex $w$ is
        redundant in $TL$, then all neighbor vertices of $w$ will
        not be searched by BFS
15:     for all vertex t in $AV_{(u,v)}(u)$ that has $\sigma(t) > \sigma(r)$
        and has been searched by the above BFS process do
16:         if $(t, d_{G'}(t,r))$ is not a redundant label entry in $SL$
            then
17:             $SL \leftarrow SL \cup (t, d_{G'}(t,r))$
18:     $SI_v \leftarrow SI_v \cup (r, SL)$
```

- Case 3: $s \in AV_{(u,v)}(u)$ and $t \in AV_{(u,v)}(u)$, or similarly, $s \in AV_{(u,v)}(v)$ and $t \in AV_{(u,v)}(v)$
- Case 4: $s \in AV_{(u,v)}(u)$ and $t \in AV_{(u,v)}(v)$, or similarly, $s \in AV_{(u,v)}(v)$ and $t \in AV_{(u,v)}(u)$

Case 1 is trivial and we must have $d_{G'}(s,t) = d_G(s,t) = dist(s,t,L)$.

In Case 2 and in Case 3, according to Lemma 6 and the definition of affected vertices (see analysis in Section 4.2), we must also have $d_{G'}(s,t) = d_G(s,t) = dist(s,t,L)$.

In Case 4, suppose $s \in AV_{(u,v)}(u)$ and $t \in AV_{(u,v)}(v)$ (the other case can be analyzed in the same way). Obviously, distance between s and t changes to a larger value due to the failed edge. If s and t become disconnected to each other in G', both will not have labels in $SI_{(u,v)}$, then we have $d_{G'}(s,t) = \infty$. If s and t is still connected in G' and without loss of generality, suppose the vertex order is $\sigma(s) < \sigma(t)$, then at least vertex t contains supplemental label entries. This is because in both the BFS AFF algorithm and the BFS ALL algorithm, the affected vertex with

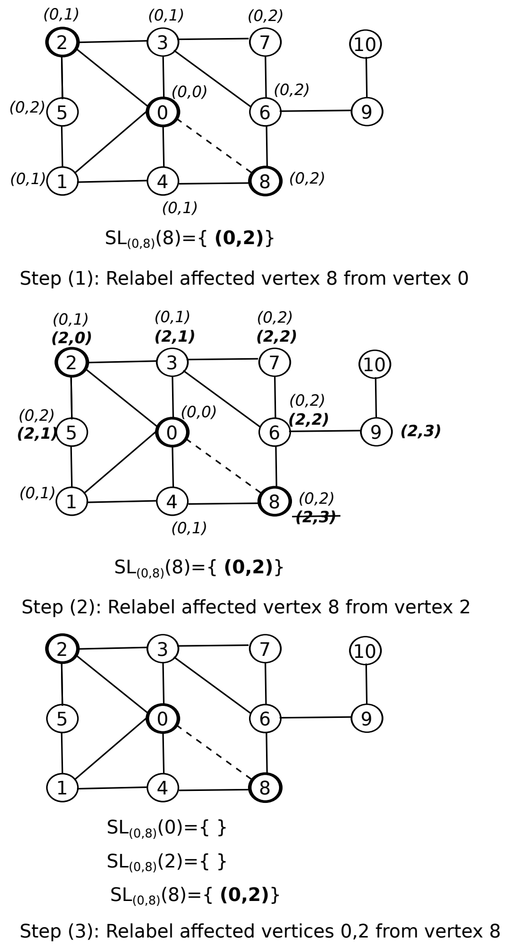


Figure 4: Supplemental index construction: BFS ALL on failed edge $(0, 8)$

minimum vertex order in $AV_{(u,v)}(u)$ (which is at most $\sigma(s)$) must produce one supplemental label entry for vertex t in $SI_{(u,v)}$ (see Lemma 3 for related details). For vertex s itself, if it does not produce any supplemental label entry for vertex t in $SI_{(u,v)}$, then it must be because the produced label entry is a redundant label. This means, in either case, the label entries of the supplemental label for vertex t in $SI_{(u,v)}$ must already contain adequate distance information for the computation of $d_{G'}(s,t)$. For example, to calculate $d_{G'}(2,8)$ in Figure 4, $SL_{(0,8)}(8) = \{(0,2)\}$ combining with $L(2) = \{(0,1)\ (2,0)\}$ in Table 1 is adequate and we can see that $d_{G'}(2,8) = 1 + 2 = 3$.

4.5 Some Remarks

Initial Index Construction. Pruned Landmark Labeling (PLL) technique presented in [2] is a state-of-the-art indexing technique for large static graphs. Indexes constructed by PLL [2] already have well-ordering property defined in Section 3. Therefore we use indexes constructed by PLL as the initial indexes for all original graphs in our experiments.

Time Complexity. Our algorithms can be directly applied on indexes constructed by PLL. Let w be the tree width [2] of G, n be the number of vertices and m be the number of edges in G. Also let (u,v) be the failed edge. If let $p = |AV_{(u,v)}(u) \cup$

$AV_{(u,v)}(v)|$ for each single-edge failure case (on average), the time complexity of the BFS AFF algorithm is $O(pn + pm)$ as it requires to perform p times BFS to compute SI_u and SI_v. Further, according to analysis of PLL in [2], the number of label entries per vertex is $O(w \log n)$. Then the time complexity of the BFS ALL algorithm is $O(nw \log n + p^2 w \log n)$, where $O(nw \log n)$ is the time upper bound to build temporary labels TL (note that p BFS rounds are enough to build the TL index that contains at most $nw \log n$ label entries) and $O(p^2 w \log n)$ is the time upper bound for redundancy tests.

5. EXPERIMENTS

We evaluated the performance of our proposed SIEF approach and this section reports the results. All experiments were performed under Linux (Ubuntu 10.04) on a server provided by eResearch SA[2]. The server was running on Dell R910 with 32 processing cores (four 8-core Intel Xeon E7-8837 CPUs at 2.67 GHz), 1024 GB main memory and 3 TB local scratch disk. All methods were implemented in C++ (the code of PLL [2] was obtained from the first author's code repository on GitHub[3]) using the same gcc compiler (version 4.4.6) with the optimizer option O3. It is worth mentioning that although we have a large amount of main memory on the server, the memory usage of our approach is in fact quite small and as observed during our experiments, the memory usage was within 12 GB for all datasets.

5.1 Datasets

Table 2 lists the six real-world datasets used in our experiments, which are briefly introduced as follows:

- `Gnutella` is a snapshot of the Gnutella peer-to-peer file sharing network collected in August 2002. Vertices represent hosts in the Gnutella network topology and edges represent connections between the hosts.
- The dataset `Facebook` consists of *circles* (or *friends lists*) from Facebook, which were collected from survey participants using a Facebook app called `Social Circles`.
- `Wiki-Vote` contains all Wikipedia voting data from the inception of Wikipedia till January 2008.
- `Oregon` is a graph of Autonomous Systems (AS) peering information inferred from Oregon route-views on May 26 2001.
- `Ca-HepTh` collaboration network of Arxiv High Energy Physics Theory category (there is an edge if authors coauthored at least one paper). The data covers papers in the period from January 1993 to April 2003 (124 months).
- `Ca-GrQc` collaboration network of Arxiv General Relativity category. Like `Ca-HepTh`, the data covers papers in the period from January 1993 to April 2003 (124 months).

More details on these datasets can be found at the Stanford Network Analysis Project website[4]. Similar to [3, 2], we treat all graphs as undirected, unweighted graphs.

[2]http://www.ersa.edu.au/

[3]https://github.com/iwiwi/pruned-landmark-labeling

[4]http://snap.stanford.edu/

It should be noted that, in Table 2, $|V|$ refers to the number of vertices and $|E|$ refers to the number of edges. In addition, IT denotes the indexing time or index construction time (in seconds) and LN denotes the average number of label entries of each vertex. We obtained these IT and LN results by using the Pruned Landmark Labeling (PLL) technique presented in [2]. As mentioned, we applied our index construction algorithms directly on the indexes constructed by PLL in our experiments.

Table 2: Real-world Datasets and Their Statistics

Dataset	|V|	|E|	IT (s)	LN
Gnutella	6,301	20,777	0.825	163.647
Facebook	4,039	88,234	0.173	25.887
Wiki-Vote	7,115	103,689	0.525	69.915
Oregon	11,174	23,409	0.080	11.189
Ca-HepTh	9,877	51,971	0.557	75.311
Ca-GrQc	5,242	28,980	0.141	43.828

5.2 Performance Evaluation

We have conducted extensive experiments to validate our proposed approach. In the experiments, we compared the numbers of affected vertices (Section 5.2.3), the average label entry numbers with and without considering edge failures (Section 5.2.1). We performed queries with and without SIEF indexes (Section 5.2.4) and great efficiency improvement was observed if using SIEF indexes. We also studied the impact of our approach in terms of index size, identification time, and relabeling time for each dataset (Section 5.2.2 to 5.2.6). Note that, we construct SIEF indexes by computing supplemental indexes for all single-edge failure cases of a given graph.

5.2.1 *Supplemental Label Entry Numbers*

Figure 5 shows the difference between the original label entry number (OLEN) without support of single-edge failures and the supplemental label entry number (SLEN) with support of single-edge failures. SLEN and OLEN of `Wiki-Vote`[5] have the biggest gap, i.e., the ratio of SLEN to OLEN is observed around 80. SLEN and OLEN of `Facebook` have the second biggest gap and the ratio of SLEN to OLEN is around 40. For other datasets, the ratios of SLEN to OLEN are all under 10. This means, compared with the total number of label entries needed for the original graphs without considering edge failures, in the case of edge failures, the total extra number of label entries (in supplemental indexes) is less than 10 times of the number of the label entries in the original index. These results indicate that the SIEF indexes are very compact.

5.2.2 *Index Size*

Figure 6 shows the original index size for the graphs with no failed edges and the supplemental index size when considering edge failures. The sum of the original index size and the supplemental index size is the total index size for handling shortest path distances on graphs with all single-edge failure cases. From the figure, the `Gnutella` dataset shows comparatively smaller proportion of its supplemental index over its total index size while the `Facebook` dataset shows

[5]We use the first three letters in the names of each dataset (e.g., `Wik` for `Wiki-vote`) for better illustration in the figure.

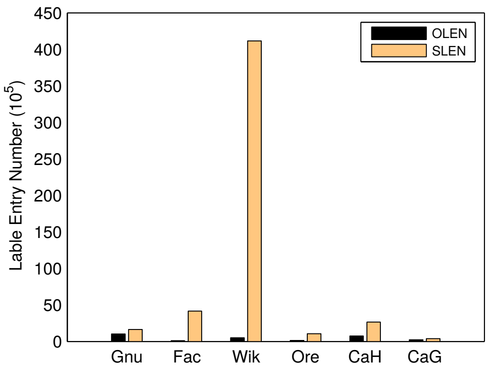


Figure 5: Comparisons between supplemental label entry numbers (SLENs) and original label entry numbers (OLENs)

largest proportion of its supplemental index over the related total index size. The `Wiki-Vote` dataset has the largest supplemental index size due to the fact that each single-failure case incurs a large number of affected vertices as well as a relatively large number of supplemental label entries (for more details, please see Table 3).

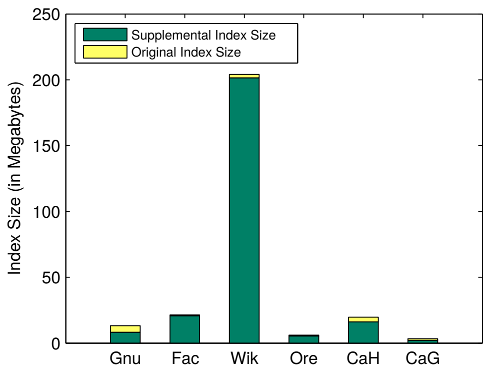


Figure 6: Index Size

5.2.3 *Affected Vertices*

Table 3 presents the relationship between affected vertices and average supplemental label entry number. *Avg* $|AU|/|V|$ represents the average percentage of affected vertices in a single-edge failure case, showing the impact of a single-edge failure on a graph. It is also the average proportion of affected vertices of the original graphs. *Avg* $|AU|$ represents the average number of affected vertices from the graph and *Avg SLEN* denotes the average number of supplemental label entries in a single-edge failure case.

From the table, we can see that the smallest percentage and the smallest average number of affected vertices are both observed in the `Ca-GrQc` dataset, with values of 1.486% and 77.884, respectively. We can also see from the table that the average supplemental label number decreases (or increases) together with the average number of the affected vertices. Also around 36% of vertices are affected in the `Wiki-Vote` dataset, which is the largest proportion. The largest average number of affected vertices is observed in the `Oregon` dataset, which is around 2,861 affected vertices for one failed edge. However, no clear linear relationship is found between the two. The largest gap occurs in the `Oregon` dataset, which indicates that the label pruning process on the affected vertices is quite powerful, leading to much fewer label entries per affected vertex. Meanwhile, the smallest gap happens in the `Gnutella` dataset and this indicates that label pruning is not very effective in this dataset.

Note that, although the proportion of affected vertices for a single-edge failure case could be large, as having been clarified in Figure 6, the final SIEF index for all single-edge failure cases is still of moderate sizes compared with the original index.

Table 3: Affected Vertices

Dataset	Avg $|AU|/|V|$	Avg $|AU|$	Avg SLEN
Gnutella	6.053%	381.386	78.445
Facebook	16.099%	650.241	47.042
Wiki-Vote	35.841%	2,550.090	396.971
Oregon	25.605%	2,861.070	45.323
Ca-HepTh	2.743%	270.881	51.095
Ca-GrQc	1.486%	77.884	13.064

5.2.4 *Query Time*

Table 4 shows the average BFS query time and the average SIEF query time. The former represents query time without using indexes proposed in this work, while the latter represents the query time when using SIEF indexes. From the table, we can see that the difference for `Oregon` dataset is the least, which still achieves at least 40 times faster when using SIEF indexes compared with the traditional BFS query approach. The largest gap occurs in the `Facebook` dataset, where the average BFS query time is around 500 times more than the SIEF query time. These results show that when using SIEF indexes, the query efficiency can be improved significantly and the query response times are normally no more than 5 μs. As mentioned in Section 4, we use supplemental indexes to support edge failures, the query process needs to examine the supplemental indexes first. When examining the supplemental indexes, SIEF checks whether the querying source and querying destination are both affected vertices given the edge failure constraint using binary search strategy. Based on the searching result, SIEF knows whether we can compute the shortest path distance based only on the supplemental indexes or based only on the original indexes. Nevertheless, the querying process is still much faster. The main reason is that the number of affected vertices for each single-edge failure case is typically small (more details are presented in Section 5.2.3) and hence the binary search process finishes quickly. This results in fast query responses in SIEF.

5.2.5 *Identification Time*

Table 5 shows the total time for identifying affected vertices for all single-edge failure cases. From the figure, we can see that, for the most datasets, the identification process can be done fairly fast and is normally finished within 80 seconds. The exception is `Wiki-Vote`, which requires a bit more than 600 seconds. The fast identification time is

Table 4: Average Query Time

Dataset	BFS Query Time	SIEF Query Time
Gnutella	140.329 μs	0.452 μs
Facebook	243.060 μs	0.522 μs
Wiki-Vote	284.867 μs	1.100 μs
Oregon	163.465 μs	4.985 μs
Ca-HepTh	325.196 μs	0.689 μs
Ca-GrQc	159.412 μs	0.479 μs

mainly because the affected vertices can be identified in a BFS manner and we only need to examine the distances between the affected vertices to one of the end vertices of a failed edge.

Table 5: Average Identification Time

Dataset	Identification Time
Gnutella	43.3708 s
Facebook	80.6844 s
Wiki-Vote	612.522 s
Oregon	35.6307 s
Ca-HepTh	36.2022 s
Ca-GrQc	4.32942 s

5.2.6 *Labeling Time*

Figure 7 shows the time for relabeling the affected vertices, which need extra distance label information to maintain correct distances to some other vertices due to a single-edge failure. Here, we used the estimated time for naive method (shown as "Estd Time for Naive Method" in the figure) as the baseline. The naive method refers to the method that we recompute a complete distance labeling index for each single-edge failure case. The process of labeling a new graph with a single-edge failure should be almost the same as the process of labeling the original graph. Therefore, the total labeling time of the naive method can be estimated by multiplying the total edge number in the original graph, i.e., the total number of single-edge failure cases, with the index time of the original graph (see Table 2).

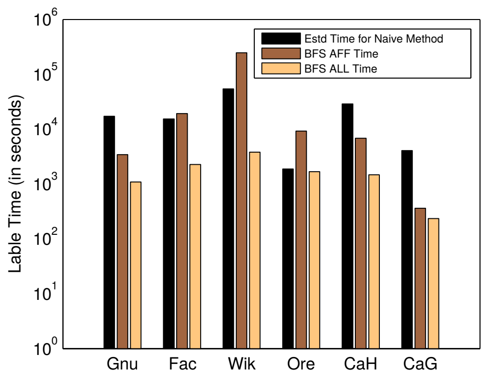


Figure 7: Labeling Time

Then, we compared the labeling times of the naive method and the two labeling methods proposed in our work: BFS AFF and BFS ALL. Recall that BFS AFF uses a late-label-pruning strategy and avoids labeling any unaffected vertices while BFS ALL uses an early-label-pruning strategy which needs labeling the unaffected vertices. From the figure, we can see that for some datasets, such as `Gnutella`, `Ca-HepTh` and `Ca-GrQc`, BFS AFF outperforms the naive method because the label-pruning process incurs some overhead when labeling unaffected vertices compared with the pure BFS process. However, BFS AFF is beaten by the naive method in terms of labeling time for other datasets, including `Facebook`, `Wiki-Vote` and `Oregon`, which contain a large number of vertices and/or a large number of edges. Hence, the late-label-pruning strategy in BFS AFF does not work well on all datasets. These results indicate that although label-pruning incurs some overhead on top of the BFS process, the label-pruning approach is quite effective in some datasets, especially datasets with more vertices and edges.

In contrast, BFS ALL performs the best on all datasets. For some datasets, such as `Facebook`, `Wiki-Vote` and `Ca-Hepth`, BFS ALL even performs orders of magnitude faster than both the naive method and the BFS AFF method. This confirms that the early-label-pruning strategy works very well on various datasets and the overhead on labeling unaffected vertices can be ignored due to the substantial label-pruning power it brings (for more details, please refer to Section 4.3).

6. CONCLUSION

This paper has studied the problem of computing the shortest path distance on graphs with single-edge failures based on 2-hop distance labeling techniques. The concept of well-ordering 2-hop distance labeling and its properties have been defined and analyzed. We have particularly focused on the constructions of compact distance labeling for all possible single-edge failure cases, a challenging problem that remains open, to the best of our knowledge. A generic framework, SIEF, has been designed for this purpose. Based on the most recent technique Pruned Landmark Labeling (PLL) [2] that handles only static graphs, we have implemented an extended version using the SIEF framework developed in this paper. Extensive experiments have also been performed on six real-world graphs to confirm its effectiveness and efficiency. SIEF is able to support compact index construction for all single-edge failure cases on graphs efficiently. Specifically, the SIEF index size is comparable to that of the indexes constructed for original static graphs, which is very compact. SIEF can answer distance queries with edge failure constraints several orders of magnitude faster than traditional Breadth-First-Search (BFS) algorithms.

In our future work, we will further investigate several aspects of answering distance queries on graphs with edge failures. The first one centers on how to support distance queries with more complex edge failure constraints, i.e., dual-failure on edges. The second aspect is to further speed up the index construction process in order to process larger graphs. Finally, it is also interesting to investigate the problem of answering distance queries on graphs with node failures, which is even more challenging than edge failures.

7. REFERENCES

[1] I. Abraham, D. Delling, A. V. Goldberg, and R. F. F. Werneck. Hierarchical Hub Labelings for Shortest

Paths. In *Proc. of the 20th Annual European Symposium on Algorithms (ESA 2012)*, pages 24–35, Ljubljana, Slovenia, 2012.

[2] T. Akiba, Y. Iwata, and Y. Yoshida. Fast Exact Shortest-Path Distance Queries on Large Networks by Pruned Landmark Labeling. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2013)*, pages 349–360, New York, NY, USA, 2013.

[3] T. Akiba, Y. Iwata, and Y. Yoshida. Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In *Proc. of the 23rd International World Wide Web Conference (WWW 2014)*, pages 237–248, Seoul, Republic of Korea, 2014.

[4] T. Akiba, C. Sommer, and K. Kawarabayashi. Shortest-Path Queries for Complex Networks: Exploiting Low Tree-Width Outside the Core. In *Proc. of the 15th International Conference on Extending Database Technology, (EDBT 2012)*, pages 144–155, Berlin, Germany, 2012.

[5] S. Baswana, U. Lath, and A. S. Mehta. Single source distance oracle for planar digraphs avoiding a failed node or link. In *Proc. of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2012)*, pages 223–232, 2012.

[6] C. Bazgan, S. Toubaline, and D. Vanderpooten. Efficient Algorithms for Finding the k Most Vital Edges for the Minimum Spanning Tree Problem. In *Proc. of the 5th International on Conference Combinatorial Optimization and Applications (COCOA 2011)*, pages 126–140, 2011.

[7] R. Bramandia, B. Choi, and W. K. Ng. Incremental Maintenance of 2-Hop Labeling of Large Graphs. *IEEE Trans. Knowl. Data Eng.*, 22(5):682–698, 2010.

[8] L. Chang, J. X. Yu, L. Qin, H. Cheng, and M. Qiao. The exact distance to destination in undirected world. *VLDB J.*, 21(6):869–888, 2012.

[9] J. Cheng, Y. Ke, S. Chu, and C. Cheng. Efficient Processing of Distance Queries in Large Graphs: A Vertex Cover Approach. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2012)*, pages 457–468, Scottsdale, AZ, USA, 2012.

[10] J. Cheng and J. X. Yu. On-line exact shortest distance query processing. In *EDBT*, pages 481–492, 2009.

[11] A. Ciortea, O. Boissier, A. Zimmermann, and A. M. Florea. Reconsidering the social web of things: position paper. In *Proc. the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp 2013) (Adjunct Publication)*, pages 1535–1544, Zurich, Switzerland, 2013.

[12] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. In *Proc. of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2002)*, pages 937–946, San Francisco, CA, USA, 2002.

[13] R. Duan and S. Pettie. Dual-failure distance and connectivity oracles. In *Proc. of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2009)*, pages 506–515, 2009.

[14] H. K. Farsani, M. A. Nematbakhsh, and G. Lausen. SRank: Shortest paths as distance between nodes of a graph with application to RDF clustering. *J. Information Science*, 39(2):198–210, 2013.

[15] A. W.-C. Fu, H. Wu, J. Cheng, and R. C.-W. Wong. IS-LABEL: an Independent-Set based Labeling Scheme for Point-to-Point Distance Querying. *Proc.of the VLDB Endowment*, 6(6):457–468, 2013.

[16] J. Hershberger and S. Suri. Vickrey Prices and Shortest Paths: What is an Edge Worth? In *Proc. of the 42nd Annual Symposium on Foundations of Computer Science (FOCS 2001)*, pages 252–259, 2001.

[17] K. Iwano and N. Katoh. Efficient Algorithms for Finding the Most Vital Edge of a Minimum Spanning Tree. *Inf. Process. Lett.*, 48(5):211–213, 1993.

[18] R. Jin, N. Ruan, Y. Xiang, and V. E. Lee. A highway-centric labeling approach for answering distance queries on large sparse graphs. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2012)*, pages 445–456, Scottsdale, AZ, USA, 2012.

[19] P. K., S. P. Kumar, and D. Damien. Ranked answer graph construction for keyword queries on RDF graphs without distance neighbourhood restriction. In *Proc. of the 20th International Conference on World Wide Web (WWW 2011, Companion Volume)*, pages 361–366, Hyderabad, India, 2011.

[20] R. Schenkel, A. Theobald, and G. Weikum. Efficient Creation and Incremental Maintenance of the HOPI Index for Complex XML Document Collections. In *Proc. of the 21st International Conference on Data Engineering (ICDE 2005)*, pages 360–371, Tokyo, Japan, 2005.

[21] S. Vassilvitskii and E. Brill. Using Web-Graph Distance for Relevance Feedback in Web Search. In *Proc. of the 29th Annual International Conference on Research and Development in Information Retrieval (SIGIR 2006)*, pages 147–153, Seattle, Washington, USA, 2006.

[22] K. Wehmuth and A. Ziviani. DACCER: Distributed Assessment of the Closeness CEntrality Ranking in complex networks. *Computer Networks*, 57(13):2536–2548, 2013.

[23] F. Wei. TEDI: efficient shortest path query answering on graphs. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2010)*, pages 99–110, Indianapolis, Indiana, USA, 2010.

[24] K. Xie, K. Deng, S. Shang, X. Zhou, and K. Zheng. Finding Alternative Shortest Paths in Spatial Networks. *ACM Trans. Database Syst.*, 37(4):29, 2012.

[25] L. Yao and Q. Z. Sheng. Exploiting Latent Relevance for Relational Learning of Ubiquitous Things. In *Proc. of the 21st ACM International Conference on Information and Knowledge Management (CIKM 2012)*, Maui, Hawaii, USA, 2012.

[26] A. D. Zhu, X. Xiao, S. Wang, and W. Lin. Efficient Single-Source Shortest Path and Distance Queries on Large Graphs. In *Proc. of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2013)*, pages 998–1006, Chicago, IL, USA, 2013.

A Selectivity based approach to Continuous Pattern Detection in Streaming Graphs

Sutanay Choudhury
Pacific Northwest National Laboratory, USA
sutanay.choudhury@pnnl.gov

Lawrence Holder
Washington State University, USA
holder@wsu.edu

George Chin
Pacific Northwest National Laboratory, USA
george.chin@pnnl.gov

Khushbu Agarwal
Pacific Northwest National Laboratory, USA
khushbu.agarwal@pnnl.gov

John Feo
Pacific Northwest National Laboratory, USA
john.feo@pnnl.gov

ABSTRACT

Cyber security is one of the most significant technical challenges in current times. Detecting adversarial activities, prevention of theft of intellectual properties and customer data is a high priority for corporations and government agencies around the world. Cyber defenders need to analyze massive-scale, high-resolution network flows to identify, categorize, and mitigate attacks involving networks spanning institutional and national boundaries. Many of the cyber attacks can be described as subgraph patterns, with prominent examples being insider infiltrations (path queries), denial of service (parallel paths) and malicious spreads (tree queries). This motivates us to explore subgraph matching on streaming graphs in a continuous setting.

The novelty of our work lies in using the subgraph distributional statistics collected from the streaming graph to determine the query processing strategy. We introduce a "Lazy Search" algorithm where the search strategy is decided on a vertex-to-vertex basis depending on the likelihood of a match in the vertex neighborhood. We also propose a metric named "Relative Selectivity" that is used to select between different query processing strategies. Our experiments performed on real online news, network traffic stream and a synthetic social network benchmark demonstrate 10-100x speedups over selectivity agnostic approaches.

1. INTRODUCTION

Social media streams and cyber data sources such as computer network traffic are prominent examples of high throughput, dynamic graphs. Application domains such as cyber security, emergency response, national security put a premium on discovering critical events as soon as they emerge in the data. Thus, processing streaming updates to a dynamic graph database for real-time situational awareness is an important research problem. These particular data sources are also distinguished by their natural representation as heterogeneous or multi-relational graphs. For example, a social media data stream contains a diverse set of entity types such as person, movie, images etc. and relations such as (*friendship, like etc.*). For cyber-security, a network traffic dataset can be modeled as a graph where vertices represent IP addresses and edges are typed by classes of network traffic [12]. Our work is focused on continuous querying of these dynamic, multi-relational graphs.

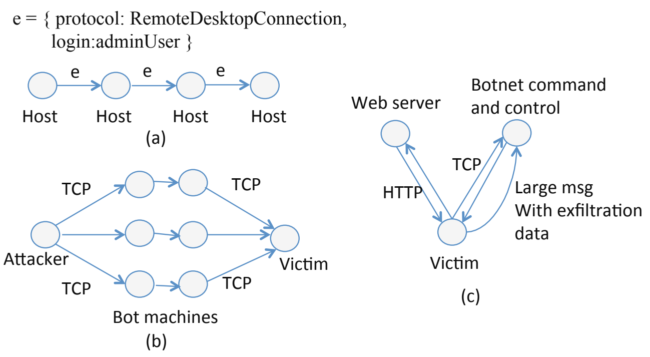

e = { protocol: RemoteDesktopConnection, login:adminUser }
e
e
e
Host
Host
Host
Host
(a)
TCP
TCP
Attacker
TCP
TCP
Victim
Bot machines
(b)
Web server
Botnet command and control
TCP
HTTP
Large msg With exfiltration data
Victim
(c)

Figure 1: Graph based descriptions of attack patterns. a) Insider infiltration: This pattern shows how an attacker may move laterally inside an enterprise, b) Denial of Service attack, c) Information exfiltration: Victim browses a compromised website. This downloads a script which establishes communication with the botnet command and control.

For social networks, we are often inundated with the stream of updates. Unless we choose to stay constantly connected to the social networks, it is highly desirable to report only the important patterns/events as they occur in the data; for example, we may choose to ask "tell me when two friends are meeting at a nearby location". The stakes are much higher in the cyber-security domain. As the volume and throughput of network traffic or event log datasets rise exponentially, the lack of ability to detect adversarial actions in real-time provides an asymmetric advantage to attackers. Internet backbone traffic collected by CAIDA [1]), which we use later as a dataset in our experiments typically accumulate 40 million packets every minute. In a study titled "Data Breach Investigations Report", US communications company Verizon analyzed 100,000 security incidents from the past decade and concluded that 90% of the incidents fell into ten attack patterns. A number of these attacks

[1]http://www.caida.org

10.5441/002/edbt.2015.15

can be naturally described as graph patterns. Figure 1 shows graph based patterns for a number of these attacks. Organizations such as internet service providers, content delivery networks etc. that receive network traffic from a wide area network are ideally poised to search for these attack patterns. Although there exists a significant number of graph databases and graph processing frameworks that scale to billion edge graphs, none of them support real-time subgraph pattern matching as a primary feature. Periodic export of network traffic flow or event alerts from log aggregation tools to a graph database, followed by post-attack querying on the static graph database is the most common workflow today. Despite cyber security being a multi-billion dollar market worldwide, the research on providing real-time querying capability on a **single, large streaming graph** is rather scarce.

Continuous querying of a dynamic graph raises a number of unique challenges. Indexing techniques that preprocess a graph and speed up queries are expensive to periodically recompute in a dynamic setting. Periodic execution of the query is an obvious solution under this condition, but the effectiveness of this approach will reduce as the interval between query executions shrinks. Also, periodic searching of the entire graph can be wasteful where the query match emerges slowly because we will find a partial match for the query every time we search and potentially redo the work numerous times. Very recent publications by Gao et al [7] and Mondal and Deshpande [15] presents algorithms for implementing continuous queries on graphs. This motivates us to study the problem of subgraph pattern matching in a streaming setting. We want to register a pattern as a graph query and continuously perform the query on the data graph as it evolves over time.

In addition to the cyber attack patterns in Figure 1, social queries are also drawn from LSBench, a benchmark for reasoning on streaming SPARQL data. A common theme that emerges is that all these query graphs are heterogenous in nature. They are composed of different edge types (in cyber security) as well as different node and edge types (in social media). None of the previous work on continuous pattern detection has addressed this issue of heterogeneity. Exploiting the heterogeneity in both the query graph and the data graph stream, and improving over heterogeneity agnostic continuous pattern detection approaches is the primary contribution of our research. The primary ideas behind our approach is described below. We believe the simplicity of our approach is its greatest strength, and it will allow easy adoption of our optimizations into the distributed system implementations developed by others in the field.

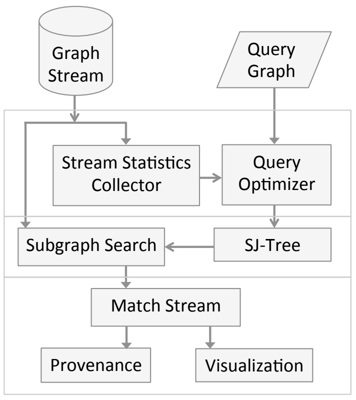


Figure 2: Framework for subgraph pattern matching on streaming graphs.

Figure 2 provides an overview of our approach. We approach the problem from an incremental processing perspective where search happens locally on every edge arrival. We do not search for the entire query graph around every new edge arriving in the stream. Given a query graph, the *query optimizer* decomposes it into smaller subgraphs as ordered by their *selectivity*. The selectivity information is obtained using the single-edge level and 2-edge path distribution obtained from the graph stream (section 5). We store the resulting decomposition into a data structure named SJ-Tree (Subgraph Join Tree) (section 3) that tracks matching subgraphs in the data graph. For a new edge in the graph, we always search for the *most selective* subgraph of the query graph. For other subgraphs of the query graph, a search is triggered if and only if a match for the previous subgraph in the selectivity order was obtained in the neighborhood of the new edge. This algorithm named "Lazy Search" is described in section 4. We introduce two metrics, *Expected and Relative Selectivity*, that captures the effectiveness of a given query decomposition (section 5). Further, we demonstrate how these metrics can be used to reason about the performance from different decompositions and select the best performing strategy.

1.1 Contributions

The most important takeaway from our work is that even as the subgraph isomorphism problem is NP-complete, it is possible to perform efficient continuous queries on dynamic graphs by exploiting the heterogeneity in the data and query graph. More specific contributions from the paper are listed below.

1. We present a dynamic graph search algorithm that demonstrates speedup of multiple orders of magnitude with respect to the state of the art.

2. We introduce two selectivity metrics for query graphs that are estimated using efficiently obtainable distributional statistics of single edge and 2-edge subgraphs from the graph stream.

3. We present an automatic query decomposition algorithm that selects the best performing strategy using the aforementioned graph stream statistics and *Relative Selectivity*.

Our observations are supported by experiments on datasets from three diverse domains (online news, computer network traffic and a social media stream).

2. BACKGROUND AND RELATED WORK

This section is aimed at providing an overview of the related field and provide the context for the studied problem. We begin with introducing the key concepts.

Multi-Relational Graphs We define a graph G as an ordered-pair $G = (V, E)$ where V is the set of vertices and the E is the set of edges that connect the vertices. In the following, we use $V(G)$ and $E(G)$ to indicate the set of vertices and edges associated with a graph G. A *labeled graph* is a six-tuple $G = (V, E, \Sigma_V, \Sigma_E, \lambda_V, \lambda_E)$, where Σ_V and Σ_E are sets of distinct labels for vertices and edges. λ_V and λ_E are vertex and edge labeling functions, i.e. $\lambda_V : V \rightarrow \Sigma_V$ and $\lambda_E : E \rightarrow \Sigma_E$.

Dynamic Graphs We define *dynamic graphs* as graphs that are changing over time through edge insertion or deletion. Every edge in a dynamic graph has a timestamp associated with it and therefore, for any subgraph g of a dynamic graph we can define a time interval $\tau(g)$ which is equal to the interval between the earliest and latest edge belonging to g. We focus on directed, labeled dynamic graphs with multi-edges in this work. The graph is maintained as a window in time. Given a time window t_W, edges are deleted as they become older than $t_{last} - t_W$, where t_{last} is the timestamp of the newest edge in the graph.

Subgraph Isomorphism Given the query graph G_q and a matching subgraph of the data graph (G_d) denoted as $G_d^{'}$, a matching between G_q and $G_d^{'}$ involves finding a bijective function $f : V(G_q) \rightarrow V(G_d^{'})$ such that for any two vertices $u_1, u_2 \in V(G_q)$, $(u_1, u_2) \in E(G_q) \Rightarrow (f(u_1), f(u_2)) \in E(G_d^{'})$.

2.1 Problem Statement

Every edge in a dynamic graph has a timestamp associated with it and therefore, for any subgraph g of a dynamic graph we can define a time duration $\tau(g)$ which is equal to the duration between the earliest and latest edge belonging to g. Given a dynamic multi-relational graph G_d, a query graph G_q and a time window t_W, we report whenever a subgraph g_d that is isomorphic to G_q appears in G_d such that $\tau(g_d) < t_W$. The isomorphic subgraphs are also referred to as *matches* in the subsequent discussions. Assume that G_d^k is the data graph at time step k. If $M(G_d^k)$ is the cumulative set of all matches discovered until time step k and E_{k+1} is the set of edges that arrive at time step $k+1$, we present an algorithm to compute a function $f\left(G_d, G_q, E_{k+1}\right)$ which returns the incremental set of matches that result from updating G_d with E_{k+1} and is equal to $M(G_d^{k+1}) - M(G_d^k)$.

2.2 Related Work

Graph querying techniques have been studied extensively in the field of pattern recognition over nearly four decades [4]. Two popular subgraph isomorphism algorithms were developed by Ullman [20] and Cordella et al. [5]. The VF2 algorithm [5] employs a filtering and verification strategy and outperforms the original algorithm by Ullman. Over the past decade, the database community has focused strongly on developing indexing and query optimization techniques to speed up the searching process. A common theme of such approaches is to index vertices based on k-hop neighborhood signatures derived from labels and other properties such as degrees and centrality [17, 18, 23]. Other major areas of work involve exploration of subgraph equivalence classes [8] and search techniques for alternative representations such as similarity search in a multi-dimensional vector space [13]. Apart from neighborhood based signatures, *graph sketches* is an important area that focuses on generating different synopses of a graph data set [22]. Development of efficient graph sketching algorithms and their applications into query estimation is expected to gain prominence in the near future.

Investigation of subgraph isomorphism for dynamic graphs did not receive much attention until recently. It introduces new algorithmic challenges because we can not afford to index a dynamic graph frequently enough for applications with real-time constraints. In fact this is a problem with searches on large static graphs as well [16]. There are two alternatives in that direction. We can search for a pattern repeatedly or we can adopt an incremental approach. The work by Fan et al. [6] presents incremental algorithms for graph pattern matching. However, their solution to subgraph isomorphism is based on the repeated search strategy. Chen et al. [2] proposed a feature structure called the *node-neighbor tree* to search multiple graph streams using a vector space approach. They relax the exact match requirement and require significant preprocessing on the graph stream. Our work is distinguished by its focus on temporal queries and handling of partial matches as they are tracked over time using a novel data structure. From a data-organization perspective, the SJ-Tree approach has similarities with the Closure-Tree [9]. However, the closure-tree approach assumes a database of independent graphs and the underlying data is not dynamic. There are strong parallels between our algorithm and the very recent work by Sun et al. [16], where they implement a query-decomposition based algorithm for searching a large static graph in a distributed environment. Here our work is distinguished by the focus on continuous queries that involves maintenance of partial matches as driven by the query decomposition structure, and optimizations for real-time query processing. Mondal and Deshpande [15] propose solutions to supporting continuous ego-centric queries in a dynamic graph, Our work focuses on subgraph isomorphism, while [15] is primarily focused on aggregate queries. We view this as complementary to our work, and it affirms our belief that continuous queries on graphs is an important problem area, and new algorithms and data structures are required for its development.

The query pattern matching approach recently proposed in [7] is most closely related to our work with some important distinctions. The authors build a vertex centric, query processing engine for dynamic graphs on top of Apache Giraph, a distributed computing framework inspired by the Pregel framework. Their query decomposition approach is based on identifying optimal sub-DAGs (directed acyclic graph) in the query graph. The DAGs' are then traversed to identify source and sink vertices to define message transition rules in the Giraph framework. Although they address significant challenges inherent of processing dynamic graphs, it is not suitable for all types of queries. Specifically, queries that have cyclic communications, such as infiltration attack query in Figure 1 cannot be decomposed in DAG to find exact matches. Also, in our work we exclusively focus on query graphs with labeled edges with specific constraints. This are not addressed in the framework proposed in [7]. Our work makes no assumptions about the query graph structure and will find exact matches even when there is no apparent sink vertices. Moreover, the focus in [7] is on distributed implementation, while we focus on selectivity based query decomposition - that can improve performance for heterogeneous graphs. We show via edge distribution and selectivity plots that real world heterogeneous graphs have a strong skew in subgraph selectivity. The novelty of our work lies in estimating the selectivity of subgraphs from the graph stream and using the selectivity to determine the subgraph search strategy.

In summary, we consider these works to pursue two related but distinct directions that needs to be implemented in a scalable system.

3. A QUERY DECOMPOSITION APPROACH

We introduce an approach that guides the search process to look for specific subgraphs of the query graph and follow *specific* transitions from small to larger matches. Following are the main intuitions that drive this approach.

1. Instead of looking for a match with the entire graph or just any edge of the query graph, partition the query graph into smaller subgraphs and search for them.

2. Track the matches with individual subgraphs and combine them to produce progressively larger matches.

3. Define a *join order* in which the individual matching subgraphs will be combined. Do not look for every possible way to combine the matching subgraphs.

Figure 3 shows an illustration of the idea. Although the current work is completely focused on temporal queries, the graph decomposition approach is suited for a broader class of applications and

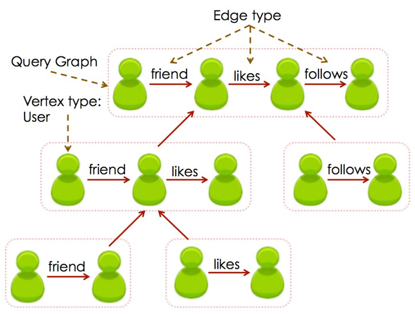


Figure 3: Illustration of the decomposition of a social query in SJ-Tree.

queries. The key aspect here is to search for substructures without incurring too much cost. Even if some subgraphs of the query graph are matched in the data, we will not attempt to assemble the matches together without following the join order.

The query decomposition approach can still suffer from having to maintain too many partial matches. If a subgraph of the query graph is highly *frequent*, we will end up tracking a large number of partial matches corresponding to that subgraph. Unless we have quantitative knowledge about how these partial matches transition into larger matches, we face the risk of tracking a large number of non-promising matching subgraphs. The “Lazy Search" approach outlined earlier in the introduction enhances this further. For any new edge, we search for a query subgraph if and only if it is the most selective subgraph in the query or if one of the either vertices in that edge participates in a match with the preceding (query) subgraph in the join order.

This section is dedicated towards introducing the data structures and algorithms for dynamic graph search. We begin with introducing the SJ-Tree structure (section 3.1) and then proceed to present the basic algorithms (Algorithm 1 and 2). The “Lazy Search"-enhanced version is introduced later in section 4. Automated generation of SJ-Tree is covered in section 5.

3.1 Subgraph Join Tree (SJ-Tree)

We introduce a tree structure called *Subgraph Join Tree (SJ-Tree)*. SJ-Tree defines the decomposition of the query graph into smaller subgraphs and is responsible for storing the partial matches to the query. Figure 3 shows the decomposition of an example query. Each of the rectangular boxes with dotted lines will be represented as a node in the SJ-Tree. The query subgraphs shown inside each “box" will be stored as a node property described below.

DEFINITION 3.1.1 A SJ-Tree T is defined as a binary tree comprised of the node set N_T. Each $n \in N_T$ corresponds to a subgraph of the query graph G_q. Let's assume V_{SG} is the set of corresponding subgraphs and $|V_{SG}| = |N_T|$. Additional properties of the SJ-Tree are defined below.

DEFINITION 3.1.2 A *Match* or a *Partial Match* is as a set of edge pairs. Each edge pair represents a mapping between an edge in a query graph and its corresponding edge in the data graph.

DEFINITION 3.1.3 Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, the join operation is defined as $G_3 = G_1 \bowtie G_2$, such that $G_3 = (V_3, E_3)$ where $V_3 = V_1 \cup V_2$ and $E_3 = E_1 \cup E_2$.

PROPERTY 1. The subgraph corresponding to the root of the SJ-Tree is isomorphic to the query graph. Thus, for $n_r = root\{T\}$, $V_{SG}\{n_r\} \equiv G_q$.

PROPERTY 2. The subgraph corresponding to any internal node of T is isomorphic to the output of the join operation between the subgraphs corresponding to its children. If n_l and n_r are the left and right child of n, then $V_{SG}\{n\} = V_{SG}\{n_l\} \bowtie V_{SG}\{n_r\}$.

Therefore, each leaf of the SJ-Tree represent subgraphs that we want to search for (perform subgraph isomorphism) on the streaming updates. Internal nodes in the SJ-Tree represents subgraphs that result from the joining of subgraphs returned by the subgraph isomorphism operations.

PROPERTY 3. Each node in the SJ-Tree maintains a set of *matches*. We define a function $matches(n)$ that for any node $n \in N_T$, returns a set of subgraphs of the data graph. If $M = matches(n)$, then $\forall G_m \in M$, $G_m \equiv V_{SG}\{n\}$.

PROPERTY 4. Each internal node n in the SJ-Tree maintains a subgraph, CUT-SUBGRAPH(n) that equals the *intersection* of the query subgraphs of its child nodes.

For any internal node $n \in N_T$ such that CUT-SUBGRAPH$(n) \neq \emptyset$, we also define a *projection operator* Π. Assume that G_1 and G_2 are isomorphic, $G_1 \equiv G_2$. Also define Φ_V and Φ_E as functions that define the bijective mapping between the vertices and edges of G_1 and G_2. Consider g_1, a subgraph of G_1: $g_1 \subseteq G_1$. Then $g_2 = \Pi(G_2, g_1)$ is a subgraph of G_2 such that $V(g_2) = \Phi_V(V(g_1))$ and $E(g_2) = \Phi_E(E(g_1))$.

Our decision to use a binary tree as opposed to an n-ary tree is influenced by the simplicity and lowering the combinatorial cost of joining matches from multiple children. With the properties of the SJ-Tree defined, we are now ready to describe the graph search algorithm.

3.2 Dynamic Graph Search Algorithm

Algorithm 1 DYNAMIC-GRAPH-SEARCH(G_d, T, edges)

```
1: $leaf\text{-}nodes$ =GET-LEAF-NODES($T$)
2: for all $e_s \in edges$ do
3:     UPDATE-GRAPH($G_d, e_s$)
4:     for all $n \in leaf\text{-}nodes$ do
5:         $g^q_{sub}$ =GET-QUERY-SUBGRAPH($T, n$)
6:         $matches$ =SUBGRAPH-ISO($G_d, g^q_{sub}, e_s$)
7:         if $matches \neq \emptyset$ then
8:             for all $m \in matches$ do
9:                 UPDATE-SJ-TREE($T, n, m$)
```

We begin with describing our dynamic graph search algorithm (Algorithm 1 and 2). The input to DYNAMIC-GRAPH-SEARCH is the dynamic graph so far G_d, the SJ-Tree (T) corresponding to the query graph and the set of incoming edges. Every incoming edge is first added to the graph (Algorithm 1, line 3). Next, we iterate over all the query subgraphs to search for matches containing the new edge (line 5-6). Any discovered match is added to the SJ-Tree (line 9).

Next, we describe the UPDATE-SJ-TREE function. Each node in the SJ-Tree maintains its sibling and parent node information (Algorithm 2, line 1-2). Also, each node in the SJ-Tree maintains a hash table (referred by the *match-tables* property in Algorithm 2, line 4). GET() and ADD() provides lookup and update operations on the hash tables. Each entry in the hash table refers to a *Match*. Whenever a new matching subgraph g is added to a node in the SJ-Tree, we compute a key using its projection ($\Pi(g)$) and insert the key and the matching subgraph into the corresponding hash table (line 12). When a new match is inserted into a leaf node we check to see if it can be combined (referred as JOIN()) with any matches that are contained in the collection maintained at its sibling node.

A successful combination of matching subgraphs between the leaf and its sibling node leads to the insertion of a larger match at the parent node. This process is repeated recursively (line 11) as long as larger matching subgraphs can be produced by moving up in the SJ-Tree. A complete match is found when two matches belonging to the children of the root node are combined successfully.

EXAMPLE Let us revisit Figure 3 for an example. Assuming we find a match with the query subgraph containing a single “friend" edge (e.g. $\{$(“George", “friend", “John")$\}$), we will probe the hash table in the leaf node with “likes" edges. If the hash table stored a subgraph such as $\{$(“John", “likes", “Santana")$\}$, the JOIN() will produce a 2-edge subgraph $\{$(“George", “friend", “John"), (“John", “likes", “Santana")$\}$. Next, it will be inserted into the parent node with 2-edges. The same process will be subsequently repeated, beginning with the probing of the hash table storing matches with subgraphs with a “follows" edge.

Algorithm 2 UPDATE-SJ-TREE($node, m$)

```
1: $sibling = sibling[node]$
2: $parent = parent[node]$
3: $k$ =GET-JOIN-KEY(CUT-SUBGRAPH[$parent$], $m$)
4: $H_s$ = match-tables[$sibling$]
5: $M_s^k$ = GET($H_s, k$)
6: for all $m_s \in M_s^k$ do
7:     $m_{sup}$ = JOIN($m_s, m$)
8:     if parent = root then
9:         PRINT('MATCH FOUND : ', $m_{sup}$)
10:    else
11:        UPDATE-SJ-TREE($parent, m_{sup}$)
12: ADD(match-tables[$node$], $k, m$)
```

4. LAZY SEARCH

Revisiting our example from Figure 3, it is reasonable to assume that the “friend" relation is highly frequent in the data. If we decomposed the query graph all the way to single edges then we will be tracking all edges that match “friend". Clearly, this is wasteful. One may suggest decomposing the query to larger subgraphs. However, it will also increase the average time incurred in performing subgraph isomorphism. Deciding the right granularity of decomposition requires significant knowledge about the dynamic graph. This motivates us to introduce a new algorithmic extension.

Assume the query graph G_q is partitioned into two subgraphs g_1 and G_q^1. We use the notation G_q^k to indicate what remains of G_q after the k-th iteration in the decomposition process. If the probability of finding a match for g_1 is less than the probability of finding a match for G_q^1, then it is always desirable to search for g_1 and look for G_q^1 only where an occurrence of g_1 is found. Therefore, we select g_1 to be the most selective edge or 2-edge subgraph in the query graph and always search for g_1 around every new edge in the graph. Once we detect subgraphs in G_d that match with g_1, we follow the same approach to search for G_q in their neighborhood. We partition G_q^1 further into two subgraphs: g_2 and G_q^2, where g_2 is another 1-edge or 2-edge subgraph.

DATA STRUCTURES With the SJ-Tree, the partitioning of G_q is done upfront at the query compile time with g_1, g_2 etc becoming the leaves of the tree. The main difference between Lazy Search and that of Algorithm 2 is that we will be searching for g_2 only around the edges in G_d where a match with g_1 is found. Therefore, for every vertex u in G_d, we need to keep track of the g_i-s such that u is present in the matching subgraph for g_i. We use a bitmap structure M_b to maintain this information. Each row in the bitmap refers to a vertex in G_d and the i-th column refers to g_i, or the i-th leaf in the SJ-Tree. If the search for subgraph g_i is enabled for vertex u in G_d, then $M_b[u][i] = 1$ and zero otherwise. Whenever a matching subgraph g' for g_i is discovered, we turn on the search for g_{i+1} for all vertices in $V(g')$. This is accomplished by setting $M_b[v][i+1] = 1$ where $v \in V(g')$.

ROBUSTNESS WITH SUBGRAPH ARRIVAL ORDER Consider a SJ-Tree with just two leaves representing query subgraphs g_1 and g_2, with g_1 representing the *more selective* left leaf. The above strategy is not robust to the arrival order of matches. Assume $g_1^{'}$ and $g_2^{'}$ are subgraphs of G_d that are isomorphic to g_1 and g_2 respectively. Together, $g_1^{'} \times g_2^{'}$ is isomorphic to the query graph G_q. Because we are searching for g_1 on every incoming edge, $g_1^{'}$ will be detected as soon as it appears in the data graph. However, we will detect $g_2^{'}$ only if appears in G_d after $g_1^{'}$. If $g_2^{'}$ appeared in G_d before $g_1^{'}$ we will not find it because we are not searching for g_2 all the time.

We introduce a small change to address this temporal ordering issue. Whenever we enable the search on a node in the data graph, we also perform a subgraph search around the node to find any match that has occurred earlier. Thus, when we find g_1 and enable the search for g_2 on every subsequent edge arrival, we also perform a search in G_d looking for g_1. This ensures that we will find g_2 even if it appeared before g_1.

Algorithm 3 summarizes the entire process. Lines 2-3 loop over all news edges arriving in the graph and update the graph. Next, given a new edge e_s, for each node in the SJ-Tree, we check to see if we should be searching for its corresponding subgraph around e_s (lines 4-8). The DISABLED() function queries the bitmap index and returns *true* if the corresponding search task is disabled. GET-QUERY-SUBGRAPH returns the query subgraph g_{sub}^q corresponding to node n in the SJ-Tree (line 9). Next, we search for g_{sub}^q using a subgraph-isomorphism routine that only searches for matches containing at least one of the end-point vertices of e_s (u and v, mentioned in line 5-6). For each matching subgraph found containing u or v, we enable the search for the query subgraph corresponding the sibling of n in the SJ-Tree. If n was not left-deep most node in the SJ-Tree, then we also query the left sibling to probe for potential join candidates (QUERY-SIBLING-JOIN(), line 16). Any resultant joins are pushed into the parent node and the entire process is recursively repeated at one level higher in the SJ-Tree.

5. SJ-TREE GENERATION

Here we address the topic of automatic generation of the SJ-Tree from a specified query graph. We begin with introducing key definitions, followed by the decomposition algorithm.

DEFINITION **Subgraph Selectivity** Given a large typed, directed graph G, the selectivity of a typed, directed subgraph g with k-edges (denoted as $S(g)$) is the ratio of the number of occurrences of g and the total number of all k-edge subgraphs in G. Instances of g may overlap with each other.

DEFINITION **Selectivity Distribution** The selectivity distribution of a set of subgraphs G_k is a vector containing the selectivity for every subgraph in G_k. The subgraphs are ordered by their frequencies in ascending order.

We present a greedy algorithm (Algorithm 4) for decomposing a query graph into its subgraphs and generating a SJ-Tree. Our choice for the greedy heuristic is motivated by extensive survey of the literature on optimal join order determination in relational databases [10, 14, 21]. A key conclusion of the survey states that *left-deep join plans* (or left deep binary trees in this case) is one

Algorithm 3 LAZY-SEARCH(G_d, T, edges)

```
1: $leaf\text{-}nodes$ =GET-LEAF-NODES($T$)
2: for all $e_s \in edges$ do
3:     UPDATE-GRAPH($G_d, e_s$)
4:     for all $n \in leaf\text{-}nodes$ do
5:         $u$ =src($e_s$)
6:         $v$ =dst($e_s$)
7:         if DISABLED(u, n) AND DISABLED(v, n) then
8:             continue
9:         $g^q_{sub}$ =GET-QUERY-SUBGRAPH($T, n$)
10:        $matches$ =SUBGRAPH-ISO($G_d, g^q_{sub}, e$)
11:        for all $m \in matches$ do
12:            if $n = 0$ then
13:                ENABLE-SEARCH-SIBLING($n, m$)
14:            else
15:                $M_j$ = QUERY-SIBLING-JOIN($n, m$)
16:                $p$ = PARENT($n$)
17:                for all $m_j \in M_j$ do
18:                    UPDATE($p, m_j$)
19:                    ENABLE-SEARCH-SIBLING($p, m$)
```

of the best performing heuristics. The above mentioned studies point to a large body of research using techniques such as dynamic programming and genetic algorithms to find the optimal join order. Nonetheless, finding the lowest cost join order or using a cost-driven join order determination remains an interesting problem in graph databases, and the approaches based on minimum spanning trees or approximate vertex cover can provide an initial path forward.

Inputs to Algorithm 4 are the query graph G_q and an ordered set of primitives M. Our goal is to decompose G_q into a collection of (possibly repeated) subgraphs chosen from M. Entries of M are sorted in ascending order of their subgraph selectivity. Given a query graph G_q, the algorithm begins with finding the subgraph with the lowest selectivity in M. This subgraph is next removed from the query graph and the nodes of the removed subgraph are pushed into a "frontier" set. We proceed by searching for the next selective subgraph that includes at least one node from the frontier set. We continue this process until the query graph is empty. SUBGRAPH-ISO performs a subgraph isomorphism operation to find an instance of g_M in G_q. Algorithm 4 uses two versions of SUBGRAPH-ISO. The first version uses three arguments, where the second argument is a vertex id v. This version of SUBGRAPH-ISO searches G_q for instances of g_M by only searching in the neighborhood of v. The other version accepting two arguments searches entire G_q for an instance of g_M. REMOVE-SUBGRAPH accepts two graphs as argument, where the second argument (g_{sub}) is a subgraph of the first graph (G_q). It removes all edges in G_q that belong to g_{sub}. A vertex is removed from G_q only when the edge removal results in a disconnected vertex.

5.1 Selectivity Estimation of Primitives

We propose computing the selectivity distribution of primitives by processing an initial set of edges from the graph stream. For experimentation purposes we assume that the selectivity order remains the same for the dynamic graph when we perform the query processing. This work does not focus on modeling the accuracy of this estimation. Modeling the impact on performance when the actual selectivity order deviates from the estimated selectivity order is an area of ongoing work.

Which subgraphs are good candidates as entries of M? Following are two desirable properties for entries in M: 1) the cost

Algorithm 4 BUILD-SJ-TREE(G_q, M)

```
1: $frontier = \emptyset$
2: while $|V(G_q)| > 0$ do
3:     $g_{sub} = \emptyset$
4:     for all $g_M \in M$ do
5:         if $frontier \neq \emptyset$ then
6:             for all $v \in frontier$ do
7:                 $g_{sub}$ =SUBGRAPH-ISO($G_q, v, g_M$)
8:                 break
9:         else
10:            $g_{sub}$ =SUBGRAPH-ISO($G_q, g_M$)
11:    if $g_{sub} \neq \emptyset$ then
12:        $frontier = frontier \cup V(g_{sub})$
13:        $G_q$ =REMOVE-SUBGRAPH($G_q, g_{sub}$)
```

for subgraph isomorphism should be low. 2) Selectivity estimation of these subgraphs should be efficient as we will need to periodically recompute the estimates from a graph stream. Based on these two criteria, we select single edge subgraphs and 2-edge paths as primitives in this study. Computing the selectivity distribution for single-edge subgraphs resolves to computing a histogram of various edge types. The selectivity distribution for 2-edge paths on a graph with V nodes, E vertices and k unique edge types can be done in $O(V(E + k^2))$ time. Algorithm 5 provides a simple algorithm to count all 2-edge paths. In our experiments, computing the path statistics for a network traffic dataset with 800K nodes and nearly 130 million edges takes about 50 seconds without any code optimization.

Algorithm 5 uses a Counter() data structure, which is a hashtable where given a key, the corresponding value indicates the number of times the key occurred in the data. A Counter() is updated via the UPDATE routine, which accepts the counter object, a key value and an integer to increment the corresponding key count. We iterate over all vertices in the input graph (G_d) (line 2). For an given vertex v, we count the number of occurrences of each unique edge type associated with it (accounting for edge directions). Line 8 iterates over all unique edge types associated with v. Next, given an edge type e_1 and its count n_1, we count the number of combinations possible with two edges of same type ($\binom{n}{2}$). Next, we compute the number of 2-edge paths that can be generated with e_1 and any other edge type e_2. We impose the LEXICALLY-GREATER constraint to ensure each edge is factored in only once in the 2-edge path distribution.

Note that we use a $Map()$ function instead of simply using the type associated with every edge. Most of our target applications have significant amount edge attributes in the graphs. As an example, in a network traffic graph we use the protocol information to determine the edge property. Thus, each network flow with the same protocol (e.g. HTTP, ICMP etc.) are mapped to the same edge type. Each flow is accompanied by multiple attributes such as source and destination ports, duration of communication etc.. Therefore, we can provide a hash function to map any user defined edge properties to an integer value. Thus, for queries with constraints on vertex and edge properties, a generic map function factors in both structural and semantic characteristics of the graph stream.

Counting the frequency for larger subgraphs is important. Given a query graph with M edges, ideally we would like to know the frequency of all subgraphs with size $1, 2, .., M - 1$. Collecting the frequency of larger subgraphs, specifically triangles have received a significant attention in the database and data mining community

[19]. Exhaustive enumeration of all the triangles can be expensive, specially in the presence of high degree vertices in the data. Approximate triangle counting via sampling for streaming and semi-streaming has been extensively studied in the recent years [11]. We foresee incorporation of such algorithms to support better query optimization capabilities for queries with triangles.

Algorithm 5 COUNT-2-EDGE-PATHS(G_d)

```
1: $P = Counter()$
2: for all $v \in V(G_d)$ do
3:     $C_v$ = $Counter()$
4:     for all $e \in Neighbors(G_d, v)$ do
5:         $e_t = Map(e)$
6:         $Update(C_v, e_t, 1)$
7:     $E_t = Keys(C_v)$
8:     for all $e_1 \in E_t$ do
9:         $n_1 = Count(C_v, e_1)$
10:        $key = (e_1, e_1)$
11:        $Update(P, key, n_1(n_1 - 1)/2)$
12:        for all $e_2 \in$LEXICALLY-GREATER$(E_t, e_1)$ do
13:            $n_2 = Count(C_v, e_2)$
14:            $key = (e_1, e_2)$
15:            $Update(P, key, n_1 n_2)$
```

5.2 Query Decomposition Strategies

Algorithm 4 shows that we can generate multiple SJ-Trees for the same G_q by selecting different primitive sets for M. We can initiate M with only 1-edge subgraphs, only 2-edge subgraphs or a mix of both. As an example, for a 4-edge query graph, the removal of the first 2-edge subgraph can leave us with 2 isolated edges in G_q. At that stage, we will create two leaf nodes in the SJ-Tree with 1-edge subgraphs. For brevity we refer to both the second and third choice as 2-edge decomposition in the remaining discussions. Clearly, these 1 or 2-edge based decomposition strategies has different performance implications. Searching for 1-edge subgraphs is extremely fast. However, we stand to pay the price with memory usage if these 1-edge subgraphs are highly frequent. On the contrary, we expect 2-edge subgraphs to be more discriminative. Thus, we will trade off lowering the memory usage by spending more time searching for larger, discriminative subgraphs on every incoming edge.

DEFINITION **Expected Selectivity** We introduce a metric called *Expected Selectivity*, denoted as $S(\hat{T_k})$. Given a SJ-Tree T_k, the Expected Selectivity is defined as the product of the selectivities of the leaf-level query subgraphs.

$leaves(T_k)$ returns the set of leaves in a SJ-Tree T_k. Given a node n, $V_{SG}(T, n)$ returns the subgraph corresponding to node n in SJ-Tree T. Finally, $S(g)$ is the selectivity of the subgraph g as defined earlier.

$$S(\hat{T_k}) = \prod_{n \in leaves(T_k)} S(V_{SG}(T_k, n)) \tag{1}$$

DEFINITION **Relative Selectivity** We introduce a metric called *Relative Selectivity*, denoted as $\xi(T_k, T_1)$. Given a 1-edge decomposition T_1 and another decomposition T_k, we define $\xi(T_k, T_1)$ as follows.

$$\xi(T_k, T_1) = \frac{S(\hat{T_k})}{S(\hat{T_1})} \tag{2}$$

We conclude the section with discussion on two desirable properties of a greedy SJ-Tree generation strategy.

THEOREM 1 Given the data graph G_d at any time t, assume that the query graph G_q is not guaranteed to be present in G_d. Then initiating the search for G_q by searching for g_{rare} where $g_{rare} \subset G_q$ and $\forall g \subset G_q ||E(g)| = |E(g_{rare})|, frequency(g) > frequency(g_{rare})$ is in optimal strategy.

PROOF The time complexity for searching for a $O(1)$ for a 1-edge subgraph and $O(\bar{d_v})$ for a 2-edge subgraph. Therefore, the runtime cost to search for g_{rare} is same as any other subgraph of G_q with the same number of edges. However, searching for g_{rare} will require minimum space because it has the minimum frequency amidst all subgraphs with same size. Therefore, searching for g_{rare} is an optimal strategy.

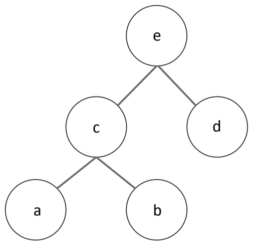


Figure 4: Example SJ-Tree used in proof of theorem 2.

THEOREM 2 Given a set of identical size subgraphs $\{g_k\}$ such that $\cup_k^n g_k = G_q$, a SJ-Tree with ordered leaves $g_k \prec g_{k+1} \prec g_{k+2}$ requires minimal space when $frequency(g_k \bowtie g_{k+1}) < frequency(g_{k+2})$.

PROOF By induction. Assume a SJ-Tree with three leaves as shown in Figure 4. Following the definitions of SJ-Tree, this is a left-deep binary tree with 3 leaves. Therefore, $frequency(c)$ denoted in shorthand as $f(c)$ $f(c)$ = $min(f(a), f(b))$. Substituting for the frequency of c, space requirement for this tree $S(T) = f(a) + f(b) + f(d) + min(f(a), f(b))$. Thus, the space requirement for this tree is minimum if $f(a) < f(b) < f(c)$.

Now we can consider any arbitrary tree where T_n refers to a tree with a left subtree T_{n_1} and a right child l_{n+2}. Above shows that T_1 constructed as above will have minimum space requirement, and so will T_2 if $f(a) < f(b) < f(c) < f(d)$.

OBSERVATION 3 Given g_k, a subgraph of query graph G_q, it is efficient to decompose g_k if there is a subgraph $g \subset g_k$, such that frequency(g) $> \left(\frac{frequency(g_k)}{d|V(g_k)|}\right)$, where $\bar{d}$ is the average vertex degree of the data graph and $|V(g_k)|$ is the number of vertices in g_k.

PROOF Given a graph g, the average cost for searching for another graph that is larger by a single edge is $\bar{d}$ multiplied by the number of vertices in g_k, and the proof follows.

Space Complexity The space complexity of the SJ-Tree can be measured in terms of the storage required by each leaf in the tree. The storage for any node in the tree is approximated by the product of the corresponding subgraph size (measured as the number of edges) and its frequency. Therefore, the space complexity of the SJ-Tree is $S(T) = \sum_k |E(g_k)| frequency(g_k)$. Given two subgraphs g_{small} and g_{big}, where g_{big} contains g_{small}, the frequency of g_{small} serves as an upper bound for g_{big}, assuming no overlapping edges. Therefore, we can assign each node in the tree to a group, where one node in each group serves to approximate the frequency of rest of the nodes in the group. Suppose $g_r(i)$ is the cardinality of the i-th group. Trivially, $\sum_i g_r(i) = N_T$, where N_T is the number of nodes in the SJ-Tree.

Therefore, given a query graph G_q and a SJ-Tree T expressing one possible query decomposition, we can estimate its space complexity as $S(T) = \sum_i g_r(i)|E(g_i)|frequency(g_i)$. There is clearly a tradeoff between the accuracy of this estimate and the computation required to obtain the necessary measurements. Approximating the space complexity in terms of single edge subgraphs is computationally easiest, although it would be a very loose bound when the frequency of a single edge subgraph is orders of magnitude higher than larger subgraphs containing that single edge subgraph. Realistically, we foresee the groups being composed of unique 1-edge, 2-edge subgraphs and triangles (if it exists in the SJ-Tree) and approximate all larger subgraph in the SJ-Tree assigned to these groups.

5.3 Comparison with selectivity agnostic approaches

Our pattern decomposition approach based on relative selectivity provides an optimal way to look for discriminate patterns compared to existing approaches. For e.g, consider the generic path query graph in 5(a). A DAG based decomposition approach [7] may look either for complete path query or decompose it randomly as shown in 5(b). As the source vertex(s1) in such a pattern may be lot more frequent than sink $v4$, our selectivity based approach will clearly identify the s2->s3->s4 pattern as being more selective and start processing search from there, clearly this is more optimal than searching for every pattern starting at s1->s2.

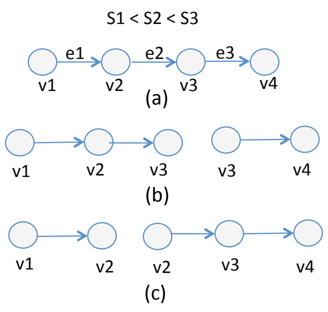


Figure 5: (a) Example path query. S_i indicates the selectivity of edge e_i. (b) A selectivity agnostic decomposition. (c) Decomposition using our selectivity based approach.

6. EXPERIMENTAL STUDIES

We perform experimental analysis on two real-world datasets (New York Times [1] (Internet Backbone Traffic data [1]) and a synthetic streaming RDF benchmark. In interest of space, we include result for CAIDA dataseti and RDF bencharк only, NYTimes performance being similar to CAIDA. The experiments are performed to answer questions in the following categories.

1. STUDYING SELECTIVITY DISTRIBUTION What does the selectivity distribution of 2-edge subgraphs look like in real world datasets? What is the duration of time for which the selectivity distribution or selectivity order of 2-edge subgraphs remains static?

2. COMPARISON BETWEEN SEARCH STRATEGIES In the previous sections, we introduced two different choices for query decomposition (1-edge vs 2-edge path based) and two different choices for query execution (lazy vs non-lazy). How do the strategies compare?

3. AUTOMATED STRATEGY SELECTION Given a dynamic graph and a query graph, can we choose an effective strategy using their statistics?

COMPARISON WITH OTHER APPROACHES Although other continuous subgraph query systems exist ([7, 15], their objectives are different. Both focus on distributed system implementations, and explore aggregate queries or approximate queries. Also, their support for the type of graph is different from ours. Our test datasets drawn from cyber security and social networks involve directed graphs with labeled vertices and edges. We believe that the research contributions complement each other; hence, we compare our implementation with a non-incremental approach that performs subgraph isomorphism for the query graph (using VF2) on every new edge in the dynamic graph. .

6.1 Experimental setup

The experiments were performed on a 32-core Linux system with 2.1 GHz AMD Opteron processors, and with 64 GB memory. The code was compiled with g++ 4.7.2 compiler with -O3 optimization.

Given a pair of data graph and query graph, we perform either of two tasks: 1) query decomposition and 2) query processing.

Query decomposition: Query decomposition involves loading the data graph, collecting 1-edge and 2-edge subgraph statistics and performing query decomposition using the selectivity distribution of the subgraphs. The SJ-Tree generated by the query decomposition algorithm is stored as an ASCII file on disk.

Query processing: The query processing step begins with loading the query graph in memory, followed by initialization of the SJ-Tree structure from the corresponding file generated in the query decomposition step. We initialize the data graph in memory with zero edges. Next, edges parsed from the raw data file are streamed into the data graph. The continuous query algorithm is invoked after each AddEdge() call to the data graph.

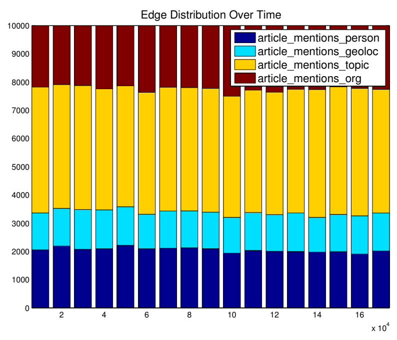
6.2 Data source description

Summaries of various datasets used in the experiments are provided in Table 1. We tested each dataset with a set of randomly generated queries. The following describes the individual datasets and test query generation.

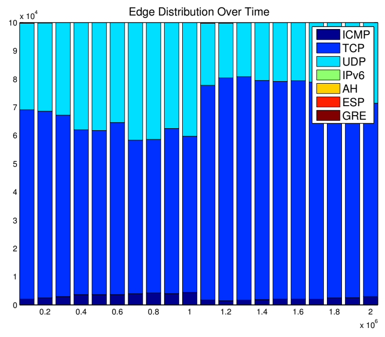
Network Traffic The dataset is an internet backbone traffic dataset obtained from `www.caida.org`. CAIDA (Cooperative Association for Internet Data Analysis) is a collaborative program that provides a wide collection of network traffic data. We used the “CAIDA Internet Anonymized Traces 2013 Dataset" for experimentation. The dataset contains 22 million network traffic flow (subsequently referred to as *netflow*) records collected over a *one minute period*. We excluded the traffic to/from IP addresses matching patterns 10.x.x.x or 192.168.x.x. These address spaces refer to private subnets and a communication from a given IP address from these spaces can actually refer to multiple physical hosts in the real word. As an example, every internet service provider configures the routers or machines inside a home network with IPs selected from the private IP address range. Therefore, if we see a request from 192.168.1.1 to google.com, there is no way to determine the exact origin of this communication. From a graph perspective, allowing private IP address and the subsequent aggregation of communication will result in the creation of vertices with giant neighbor lists, which will surely impact the search performance. A detailed list of use cases describing subgraph queries for cyber traffic monitoring are described in [12].

Social Media Stream Our final test dataset is a synthetic RDF social media stream available from the Linked Stream Benchmark

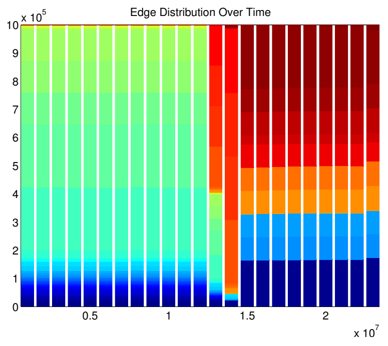
[1]http://data.nytimes.com
[1]http://www.caida.org



(a) Online news - New York Times



(b) Internet Backbone Traffic - CAIDA



(c) Synthetic social data stream in RDF

Figure 6: Edge type distribution shown with the evolution of the dynamic graph.

Table 1: Summary of test datasets

Dataset	Type	Vertices	Edges
Internet Backbone Traffic	Network traffic	2,491,915	19,550,863
LSBench/CSPARQL Benchmark	RDF Stream	5,210,099	23,320,426
New York Times	Online News	64,639	157,019

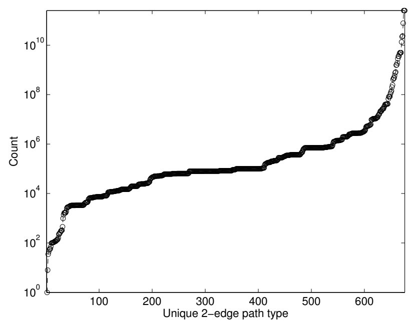
(LSBench) [1]. We generated the dataset using the sibgenerator utility with 1 million users specified as the input parameter. The generated graph has a static and a streaming component. The static component refers to the social network with user profiles and social network relationships. The streaming component includes 3 streams. The *GPS stream* includes user checkins at various locations. The *Post and Comments stream* includes posts and comments by the users, subscriptions by users to forums, and a stream of "likes" and "tags". Finally, the *photo stream* includes information about photos uploaded by users, and "tags" and "likes" as applied to photos.

6.3 Selectivity Distribution

Figure 6 shows the edge distribution plotted over time. X-axis shows the number of cumulative edges in the graph as it is growing. The plotted distribution is not cumulative. The edge distribution is collected after fixed intervals. The interval is 10 thousand, 100 thousand and 1 million respectively. There are 4, 7, and 45 edge types in these datasets. The first half of the RDF dataset contains data for a simulated social network. The second half contains simulated data about the activities in the network such as posts, and checkins at locations The shift in the edge distribution around the mid point reflects these different characteristics. The key observation is that the relative order of different types of edges stays similar even as the graph evolves.

There were 14, 62 and 676 unique 2-edge paths present in the New York Times, netflow and LSBench datasets. Figure 7 shows the 2-edge path distribution for the LSBench dataset. We found a small number of 2-edge subgraphs to dominate the distribution across all the datasets. Other datasets show a similarly skewed distribution, and was omitted for space. The skew is heaviest for the LSBench dataset, which is expected given the higher number of unique edge types and the larger size of the dataset.

The goal of this analysis was to observe the variability in the selectivity distribution over time. The selectivity distribution is expected to vary over time. However, it is the relative order of the unique single edge or 2-edge subgraphs that matters from the query decomposition perspective. For each of the test datasets, we took



(a) Synthetic social data stream in RDF

Figure 7: 2-edge path distribution in each test data set. Each point on X-axis represents a unique 2-edge path and Y-axis shows its corresponding count.

multiple snapshots of the selectivity order and found it to be stable, except with fluctuations for the very low frequency components (data points on the left end of the distributions in Fig. 7). Significant changes in the selectivity order can adversely impact the performance of the query. Estimating the duration over which the selectivity ordering stays stable for a given data stream, quantification of errors based on shift in the distribution, and adapting the query algorithm to handle such shifts is reserved for future work.

6.4 Query Performance Analysis

This section presents query performance results obtained through query sweeps on the network traffic and social network dataset. We restrict the analysis to these two datasets for their larger size. The analysis on New York Times dataset made available in the Appendix section in the interest of space. For each query, we collect performance from 4 different query execution strategies obtained by 1-edge or 2-edge decomposition of a query graph and the lazy vs. track everything approach adapted by the query algorithm. The following tags are used to describe the plots in the remainder of the paper: a) "*Single*": 1-edge decomposition, search tracks all matching subgraphs in SJ-tree, b) "*SingleLazy*": 1-edge based query decomposition, use "Lazy" approach to search, c) "*Path*": 2-edge decomposition, search tracks all matching subgraphs in SJ-Tree, and d) "*PathLazy*": 2-edge decomposition with "Lazy" search.

6.4.1 Network Traffic and LSBench

We present aggregated results for each query group for LSBench and CAIDA. Both of these datasets are orders of magnitude larger than New York Times and the scale allows us to magnify the differences between multiple strategies.

QUERY GENERATION We generate both path queries and binary tree queries for the netflow data. Figure 8 shows two decompositions of an example query. The vertex labels are fixed to type "ip" and the edge types are randomly chosen from a set of 7 protocols: ICMP, TCP, UDP, IPv6, AH, ESP and GRE. The binary tree queries were generated following the test generation methodology described in [16]. The LSBench dataset is tested with path queries and n-ary trees. A list of valid triples (vertex type, edge type , vertex type) is generated using the LSBench schema. A tree query is generated by randomly selecting an edge from the set of valid triples and then iteratively adding valid new edges from any of the nodes available. All our query graphs are unlabeled. Using netflow data as an example, we do not generate a query that has a label associated with any of the nodes. In practice, we expect users to employ labeled queries such as finding a tree pattern in the network traffic where the root of the tree has a IP address (i.e. label) from a certain subnet. For social data, we may look for paths with specified user ids (node labels) on the source and the destination nodes on the path. Here, our experiments are motivated to study the impact of subgraph distributional statistics on query processing.

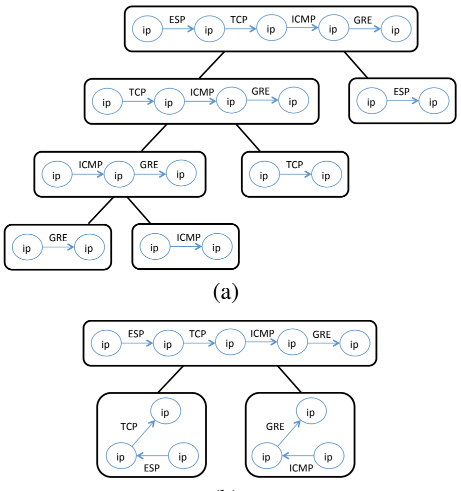


Figure 8: 1 and 2-edge based decompositions of a path query on netflow traffic data.

COMPARISON WITH OTHERS In our previous work [3] we had compared the performance of our implementation with the IncIsoMatch algorithm proposed by Fan et al. [6]. Our IncIsoMatch implementation was based on a variant of the well-known VF2 algorithm [5].

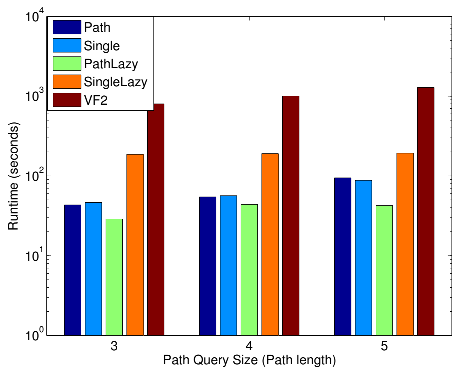
SUMMARIZATION OF RESULTS All queries of the same type (path or tree) and size (3-hop length or 5 nodes) are denoted as a group. We generated 100 queries for each group and then eliminated ones that contained 2-edge paths not seen in the sampled path distribution. This was done for two reasons; first, inclusion of an unseen 2-edge path combination makes the query artificially discriminative. Our goal is to observe query processing time as a function of varying selectivity, so including unusually discriminative queries bias our studies. Second, when asked to generate a path-based decomposition, our SJ-Tree generator resorts to generating a single-edge based decomposition when a query subgraph contains an unseen 2-edge path. This would bias our comparison between a path-based decomposition and single-edge based decomposition. Finally, for all the "valid" queries we further sampled them by the Expected Selectivity computed using 2-edge path distribution and reduced each group to a smaller set of queries that provide a near uniform sampling of the Expected Selectivity from the larger set. Finally, the reported runtime for a given strategy (e.g. "PathLazy") is obtained by averaging the runtimes from the reduced set of queries,

Figure 9a-d shows the query processing times collected for both datasets. The size of the query processing window was fixed at 8M triples, and the performance statistics were collected at at the middle and at the end of the graph stream. We profiled different components of the query processing such as the time spent in performing subgraph isomorphism and the time spent in updating the SJ-Tree. The latter is largely composed of the time spent in looking up the hash tables in various nodes of the SJ-Tree, performing joins between partial matches and inserting new entries. We found that the subgraph isomorphism operation (for 1 or 2-edge subgraphs) dominates the processing time. Considering both classes of queries with diameter 4 and 5, the subgraph isomorphism operation consumes more than 95% of the total query processing time.

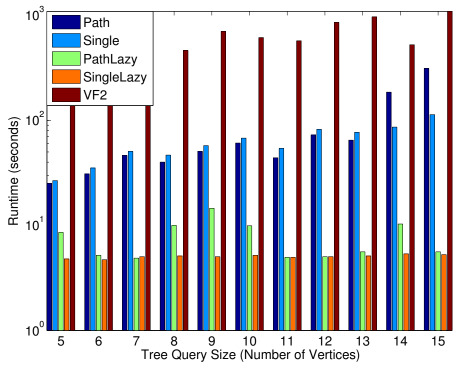
A general observation is that the performance of non-incremental search by VF2 is found to be 10-100x slower. The Y-axis is plotted in log scale, and we can see how the run times of the "Path" and "Single" approaches rise exponentially as the query sizes are increased. Overall, we find the "SingleLazy" and "PathLazy" are the best performing search approaches. As the tree queries show, the growth rate in the query processing time is much slower for the "Lazy" variants. This conclusively demonstrates the effectiveness of restricting the search to where a match is emerging, and growing the match by starting from the most selective sub-query.

6.5 Analysis via Relative Selectivity

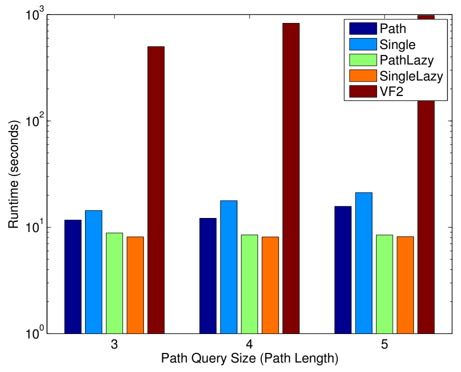
Figure 10 shows the distribution of relative selectivity for queries with 4 edges across all three datasets. We picked query graphs with 4 edges to find a common basis for comparing different type of queries (k-partite vs. path queries) across multiple datasets, and the discussion is equally applicable to larger or different query class combinations. The top subplot shows the relative selectivity of 10 k-partite queries from the New York Times data. For netflow and LSBench, we randomly sampled 25 queries from the randomly generated path query collection. As can be seen, the relative selectivity is very low for the netflow dataset. Following the definition of relative selectivity, its value is lowered when the path distribution based selectivity is low. In other words, there are some paths in the query which have very low probability of occurrence. Therefore, the "PathLazy" approach is superior for such queries. Empirical observation on larger path queries and other tree queries seem to suggest two prominent clusters of relative selectivity values. The first one typically ranges from 0.001 and above, and the second one contains values that are smaller by multiple orders of magnitude. This suggests a heuristic that "PathLazy" strategy could be employed for queries with relative selectivity below 0.001, and "SingleLazy" be employed for queries above 0.001.



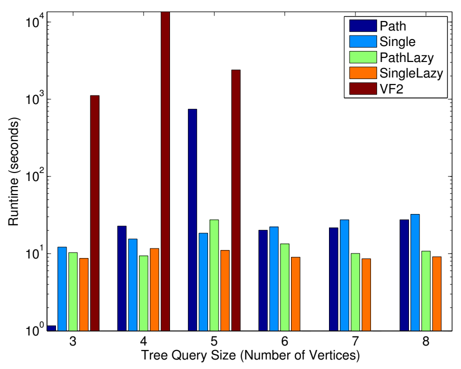
(a) Runtime for Path Queries on Netflow data.



(b) Runtime for Tree Queries on Netflow data.



(c) Runtime for Path Queries on LSBench data.



(d) Runtime for Tree Queries on LSBench data.

Figure 9: Runtimes from Path and Tree Queries on Netflow and LSBench.

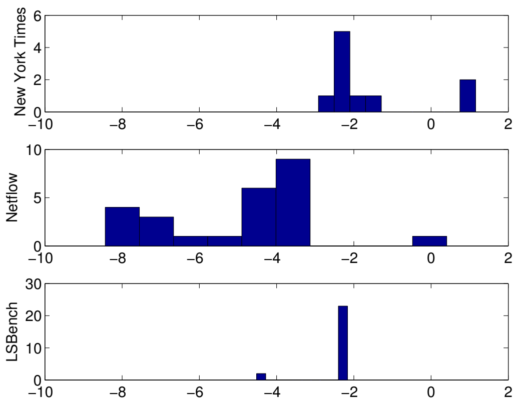


Figure 10: Distribution of Relative Selectivity across queries with 4 edges in 3 datasets. Relative selectivity is shown on X-axis in log scale.

7. CONCLUSION AND FUTURE WORK

We present a new subgraph isomorphism algorithm for dynamic graph search. We analyzed multiple real-world datasets and discovered that the distribution of 2-edge subgraphs are heavily skewed. We further demonstrated with a “Lazy" search algorithm that a query decomposition strategy exploiting this skew will be consistently efficient. Finally, we concluded with a Relative Selectivity based rule for selecting a search strategy.

The problem of continuous pattern detection is an emerging area, and there is an open field to explore. While our 2-edge subgraph based approach provides an initial foundation, deeper investigations are warranted for more accurate selectivity estimation. Subsequent research can leverage on the significant body of work on counting larger subgraphs such as triangles in streaming or semi-streaming scenarios to obtain quantitative estimates of space complexity of a given query decomposition. Adaptive query processing is an important follow-up problem as well. A long standing database query needs to be robust against shift in the data characteristics. While we propose a fast algorithm for periodic recomputation of the primitive distribution, we do not address the issues of modeling the inefficiency from operating under a different selectivity order and migrating existing partial matches from one SJ-Tree to another.

8. REFERENCES

[1] D. F. Barbieri, D. Braga, S. Ceri, E. D. Valle, and M. Grossniklaus. C-sparql: a continuous query language for rdf data streams. *Int. J. Semantic Computing*, 4(1), 2010.

[2] L. Chen and C. Wang. Continuous subgraph pattern search over certain and uncertain graph streams. *IEEE Trans. on Knowl. and Data Eng.*, 22(8):1093–1109, Aug. 2010.

[3] S. Choudhury, L. Holder, G. Chin, and J. Feo. Fast search for multi-relational graphs. *ACM SIGMOD Workshop on Dynamic Network Management and Mining*, 2013.

[4] D. Conte, P. Foggia, C. Sansone, and M. Vento. Thirty years of graph matching in pattern recognition. *Intl. Journal of Pattern Recognition and Artificial Intelligence*, 2004.

[5] L. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE Trans. on Pattern Analysis and Machine Intelli.*, 2004.

[6] W. Fan, J. Li, J. Luo, Z. Tan, X. Wang, and Y. Wu. Incremental graph pattern matching. SIGMOD '11, 2011.

[7] J. Gao, C. Zhou, J. Zhou, and J. Yu. Continuous pattern detection over billion-edge graph using distributed framework. In *2014 IEEE 30th International Conference on Data Engineering (ICDE)*, 2014.

[8] W.-S. Han, J. Lee, and J.-H. Lee. Turboiso: Towards ultrafast and robust subgraph isomorphism search in large graph databases. SIGMOD '13.

[9] H. He and A. K. Singh. Closure-tree: An index structure for graph queries. ICDE '06.

[10] J. M. Hellerstein and M. Stonebraker. *Predicate migration: Optimizing queries with expensive predicates*, volume 22. ACM, 1993.

[11] M. Jha, C. Seshadhri, and A. Pinar. A space efficient streaming algorithm for triangle counting using the birthday paradox. In *SIGKDD*. ACM, 2013.

[12] C. Joslyn, S. Choudhury, D. Haglin, B. Howe, B. Nickless, and B. Olsen. Massive scale cyber traffic analysis: a driver for graph database research. In *1st ACM SIGMOD Workshop on Graph Data Management Experiences and Systems*, 2013.

[13] A. Khan, N. Li, X. Yan, Z. Guan, S. Chakraborty, and S. Tao. Neighborhood based fast graph search in large networks. SIGMOD '11.

[14] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *VLDB*, volume 86, pages 128–137. Citeseer, 1986.

[15] J. Mondal and A. Deshpande. Eagr: Supporting continuous ego-centric aggregate queries over large dynamic graphs. In *SIGMOD 2014*.

[16] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *PVLDB*, 5(9), 2012.

[17] Y. Tian and J. Patel. Tale: A tool for approximate large graph matching. In *ICDE '08*.

[18] H. Tong, C. Faloutsos, B. Gallagher, and T. Eliassi-Rad. Fast best-effort pattern matching in large attributed graphs. KDD '07.

[19] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos. Doulion: counting triangles in massive graphs with a coin. In *SIGKDD*, 2009.

[20] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23:31–42, January 1976.

[21] Y. Wu, J. M. Patel, and H. Jagadish. Structural join order selection for xml query optimization. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 443–454. IEEE, 2003.

[22] P. Zhao, C. C. Aggarwal, and M. Wang. gsketch: on query estimation in graph streams. *PVLDB*, 5(3), 2011.

[23] P. Zhao and J. Han. On graph query optimization in large networks. *PVLDB.*, 3:340–351, September 2010.

APPENDIX

A. ANALYSIS OF DYNAMIC GRAPH SEARCH ALGORITHM

At this point, it is probably obvious that different SJ-Tree structures can be generated from the same query graph (Figure 8). While multiple factors can lead to generation of different SJ-Trees, one primary factor is our choice for granularity of decomposition, the size and the structure of the subgraphs we decompose the query to. Henceforth, we often refer to these set of small subgraphs as *search primitives* or simply *primitives.* As a first step to understand the speed-memory tradeoff associated with different choices for primitives, we begin with the complexity analysis of the dynamic graph search described in Algorithm 1 and 2. A key operation in Algorithm 1 is the process of subgraph isomorphism around every new edge in the graph. Therefore, we exclusively focus on the complexity analysis in terms of 1-3 edge subgraphs as candidates for search primitives.

SINGLE EDGE SUBGRAPHS When the query graph (g^q_{sub} in Algorithm 1, line 5) contains a single edge, checking if an edge from the data graph (e_s) matches the query edge require comparing the types and potentially other attributes of the edges. Depending on the query constraint, we may need to look up the node label to perform a string comparison or evaluate a regular expression. The node labels or any other node-specific properties are stored in an array leading to constant time access to node labels. Therefore, a single-edge query can be matched in $O(1)$ time.

TRIADS Assume that the query graph is a triad with three vertices v_1, v_2 and v_3, and edges ordered as $e_1 = (v_1, v_2), e_2 = (v_2, v_3), e_3 = (v_3, v_1)$. For any edge e in the data graph, we can detect a match with e_1 in constant time. If e is matched, we search the neighborhood of the vertex that matches with v_2 to search for e_2. Denoting this vertex as $v_2^{'}$, the cost of this second level of search is $O(degree(v_2^{'}))$. In case of a 3-edge subgraph, each of the successful second level searches proceed to find a match for the third edge. Thus, the cost of a 2-edge subgraph is $O(degree(v_2^{'}))$ and a 3-edge subgraph is $O(degree(v_2^{'}) * degree(v_3^{'}))$. We can refine these estimates to obtain an average cost of the search as $O(\bar{d_2})$ for a 2-edge subgraph and $O(\bar{d_2}\bar{d_3})$ for a 3-edge subgraph, where $\bar{d_2}$ and $\bar{d_3}$ are the average degree of the vertices in the graph for the types of v_2 and v_3.

The next step is to estimate a cost for the SJ-Tree update operation (Algorithm 2). We begin with the hash-join operation (Algorithm 2, line 7). Assume the frequency of a graph g^i_q is n_i, where the frequency of a subgraph is defined as the count of its instances over an edge stream of length N. Therefore, over N edges, we can expect $O(n_1)$ matches for g^1_q and $O(n_2)$ matches for g^2_q. Therefore, H_2 (hash table associated with the SJ-Tree node representing g^2_q) will be probed for a match $O(n_1)$ times over N edges and H_1 (associated with the SJ-Tree node representing g^1_q) will be probed $O(n_2)$ times within the same period.

If we knew the frequency of G_q, henceforth referred as $f_S(G_q)$, then we can also estimate the number of new subgraphs that will be produced as the result of the hash-joins. Given that the frequency of the larger subgraph can not exceed that of the more selective component we can approximate $O(n(G^q)) \simeq min\left(O(n_1), O(n_2)\right)$. Therefore, the average work for every incoming edge in the graph can be expressed as,

$$\left(f_S(g^1_q) + f_S(g^2_q) + O(n_1) + O(n_2) + min\left(O(n_1), O(n_2)\right)\right)/N.$$

The Hash-Join combined with leaf level searches provides the simplest example of a SJ-Tree, a binary tree with height 1. In this section, we analyze the time complexity of the query processing as it happens in a multi-level SJ-Tree. Given any non-leaf node n, we can obtain the expression for average work by adapting the complexity expression shown above. Note that if a child of n, denoted by n_c, is not a leaf level node but an internal node, then the term corresponding to the search cost ($f_S(g)$) disappears. Additionally, we can replace the search cost with the cost corresponding to the average work incurred by the subtree rooted by n_c. Therefore, given a SJ-Tree (T_{sj}) the average work ($C(T_{sj})$) can be obtained by recursive computation from the root. $C(T_{sj}) = C(root(T_{SJ}))$

Scaling Unbound-Property Queries on Big RDF Data Warehouses using MapReduce[*]

Padmashree Ravindra
Department of Computer Science
North Carolina State University
pravind2@ncsu.edu

Kemafor Anyanwu
Department of Computer Science
North Carolina State University
kogan@ncsu.edu

ABSTRACT

Semantic Web technologies are increasingly at the heart of many integrated scientific and general purpose data warehouses. Flexible querying of such diverse data collections with (partially) unknown structures can be enabled using triple patterns with 'unbound' properties (edges with don't care labels). When evaluating such queries using relational joins, intermediate results contain redundancy due to repeated combination of bound-property mappings with those of the unbound properties. However, in distributed-processing contexts, the footprint of intermediate results directly impacts I/O and communication costs. Given the popularity of MapReduce-based platforms for periodic on-demand scaling using Cloud resources, we propose an algebraic optimization technique that interprets unbound-property queries on MapReduce, using a non-relational algebra based on a TripleGroup data model. The approach enables shorter execution workflows and reduced costs for processing RDF queries on MapReduce. This paper introduces new logical and physical operators, and query rewriting rules for interpreting unbound-property queries using the TripleGroup-based data model and algebra. A key optimization strategy is to concisely represent intermediate results as far along an execution workflow as possible, thus minimizing the effects of redundancy. The proposed work is integrated into Apache Pig. Experiments conducted on real-world and synthetic benchmark datasets demonstrate their benefit over popular relational-style MapReduce systems.

1. INTRODUCTION

The successful adoption of Semantic Web technologies to interlink diverse (related) datasets has led to large semantically-integrated scientific (Uniprot [8], Bio2RDF [9]) and general purpose (DBpedia [7], Billion Triple Challenge [1]) RDF data warehouses. The heterogeneous and evolving nature of such data collections makes it difficult for users to be familiar with different kinds of relationships that exist in the data. Consequently, exploration of datasets in data-integration [23] and data archival [36] scenarios require flexibility in querying, i.e., the ability to use structural variables or "don't cares" in queries. SPARQL [28], the standard query language to specify graph pattern queries on the Semantic Web, enables flexible querying of datasets by allowing `OPTIONAL` substructures or substructures with missing edge labels. The latter are called *unbound-property* triple patterns and can be used to query unknown relationships ("*Scientists in some way associated to the same city*"), relationships with partial knowledge ("*Gene Ontology terms related to a gene Rxr*"), or to retrieve all available information about a resource ("*What is known about the Hexokinase gene?*").

Consider an example SPARQL query $Q1$ on Bio2RDF, a Life Sciences RDF dataset. $Q1$ is useful to analyse the Parkinson's disease and involves two unbound-property triple patterns (1) and (5).

Query Q1	**Description**
`SELECT` ?s1, ?label1, ?s2, ?label2, ?o2	*Retrieve gene ontology (GO) terms*
`WHERE` {	*related to "rxr", a gene of interest*
?s1 ?p1 ?o1 . (1)	*in analyzing Parkinson's disease.*
`FILTER` regex(?o1, "rxr")	*Q1 contains two star subpatterns,*
?s1 label ?label1 . (2)	*SJ_1 (1-2) and SJ_2 (3-5).*
?s2 xGO ?o2 . (3)	*(1) matches triples whose object*
?s2 label ?label2 . (4)	*contains string "rxr" (any property).*
?s2 ?p2 ?s1 . (5)	*(5) specifies an unknown*
}	*relationship connecting the two*
	star subpatterns in the query.

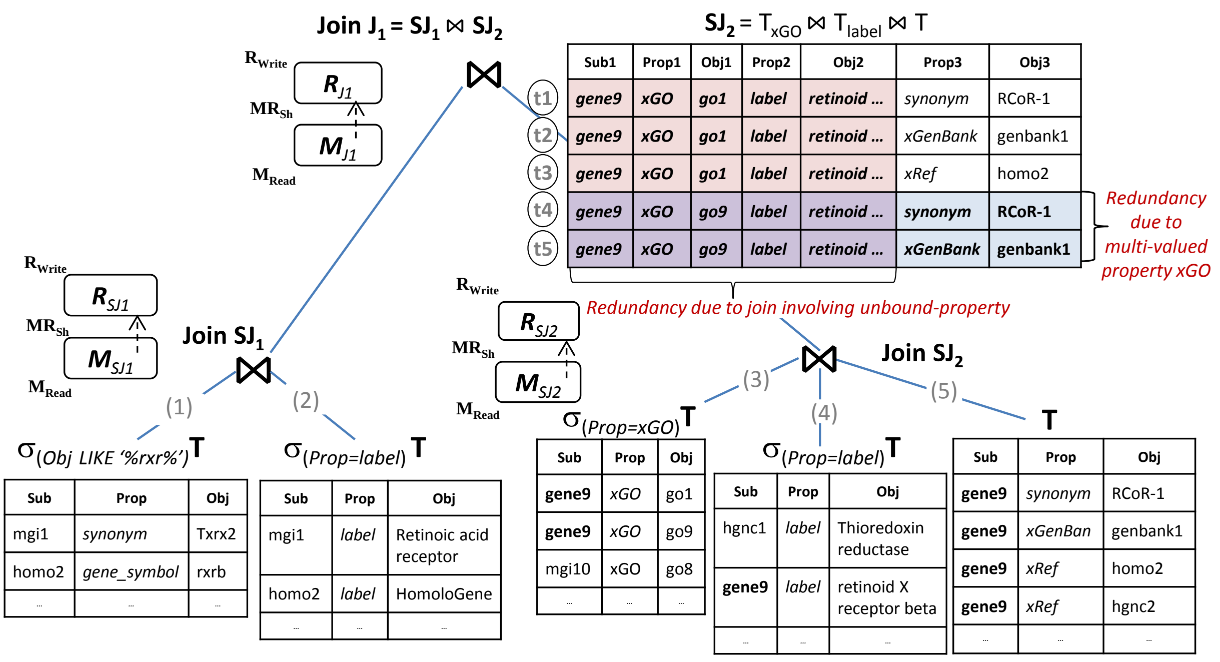
Other than querying scenarios in integrated data warehouses, subqueries with unbound-property triple patterns are also generated while optimizing ontological queries by rewriting them as a union of conjunctive queries. Examples of unbound-property queries can be found in real [23] and synthetic Semantic Web benchmarks [11], as well as other studies [22, 36]. In fact, 84% of queries in [2] involve unbound-property triple patterns.

Given a triple relation T and subset relations T_{xGO} and T_{label} with property types xGO and $label$, respectively, the subquery SJ_2 can be evaluated using relational joins ($T_{xGO} \bowtie T_{label} \bowtie T$). Figure 1 (right) shows the subrelations of T participating in SJ_2 and a snapshot of the star-join result. An issue with intermediate results in such cases is redundancy. For example, the result for SJ_2 in Figure 1(top right) contains repeated occurrences for matches of the bound properties – xGO and $label$, with each match of the unbound-property triple pattern. The numbers of matches for the unbound-property triple pattern could be large if properties in the input dataset have high multiplicity ($gene9$ is associated with multiple $xRef$), further aggravating the issue of redundancy. High-multiplicity properties are common in real-world social networks as well as biological datasets such as Uniprot and Bio2RDF, e.g., some Uniprot properties have multiplicity as high as 13K.

For applications with periodic scale–up requirements, the growing trend is to employ cloud-processing platforms, e.g., Hadoop [10], Dryad [16], Hive [37], Pig [26], that are based on the MapReduce [12] computing model. However, any redundancy in interme-

[*]The work presented in this paper is partially funded by NSF grant IIS-1218277.

10.5441/002/edbt.2015.16


Join $J_1 = SJ_1 \bowtie SJ_2$
R_{Write}
R_{J1}
MR_{Sh}
M_{J1}
M_{Read}
$SJ_2 = T_{xGO} \bowtie T_{label} \bowtie T$
t1
t2
t3
t4
t5
Redundancy due to multi-valued property xGO
Redundancy due to join involving unbound-property
R_{Write}
R_{SJ1}
MR_{Sh}
M_{SJ1}
M_{Read}
Join SJ_1
(1)
(2)
R_{Write}
R_{SJ2}
MR_{Sh}
M_{SJ2}
M_{Read}
Join SJ_2
(3)
(4)
(5)
$\sigma_{(Obj\ LIKE\ '\%rxr\%')}$ T
$\sigma_{(Prop=label)}$ T
$\sigma_{(Prop=xGO)}$ T
$\sigma_{(Prop=label)}$ T
T

Sub1	Prop1	Obj1	Prop2	Obj2	Prop3	Obj3
gene9	***xGO***	***go1***	***label***	***retinoid …***	*synonym*	RCoR-1
gene9	***xGO***	***go1***	***label***	***retinoid …***	*xGenBank*	genbank1
gene9	***xGO***	***go1***	***label***	***retinoid …***	*xRef*	homo2
gene9	***xGO***	***go9***	***label***	***retinoid …***	***synonym***	**RCoR-1**
gene9	***xGO***	***go9***	***label***	***retinoid …***	***xGenBank***	**genbank1**

Sub	Prop	Obj
mgi1	*synonym*	Txrx2
homo2	*gene_symbol*	rxrb
…	…	…

Sub	Prop	Obj
mgi1	*label*	Retinoic acid receptor
homo2	*label*	HomoloGene
…	…	…

Sub	Prop	Obj
gene9	*xGO*	go1
gene9	*xGO*	go9
mgi10	xGO	go8
…	…	…

Sub	Prop	Obj
hgnc1	*label*	Thioredoxin reductase
gene9	*label*	retinoid X receptor beta
…	…	…

Sub	Prop	Obj
gene9	*synonym*	RCoR-1
gene9	*xGenBan*	genbank1
gene9	*xRef*	homo2
gene9	*xRef*	hgnc2
…	…	…

Figure 1: A MapReduce workflow for an unbound-property graph pattern query $Q1$ with two star subqueries SJ_1 and SJ_2; Join result of unbound-property star subpattern SJ_2 contains redundant information related to bound properties (xGO, label)

diate results impacts query processing costs, particularly for MapReduce based distributed processing platforms that involve shipping of intermediate results across the network. The intermediate result footprint also impacts additional costs associated with sorting phases, materialization between the 2-steps of a MapReduce (MR) execution cycle, and total disk space requirements to store all intermediate states for fault-tolerance purposes. Hence, it is critical to minimize the footprint of intermediate results.

1.1 Related Work

Optimizing Relational Query Plans on MapReduce: There have been several efforts to shorten the length of MR workflows [6, 40, 15, 5, 27] to minimize the overall costs of MapReduce-based processing, sharing scans [24, 25, 39] and computations [24, 13] across MR workflows, cost-based and transformation-based MR workflow optimizer [20], and data skew problems [19]. Multi-way join algorithms [6, 40] cluster multiple joins into a single [6] or few [40] MR cycles, but have not been applied to join-intensive workloads. Amongst the MapReduce-based RDF processing systems, SHARD [32] uses initial MR cycles to cluster triples into star subgraphs, followed by separate MR cycles to process each clause in the SPARQL query. HadoopRDF [15] pre-processes triples using the vertical-partitioning (VP) [4] approach, and uses heuristics to greedily group non-conflicting joins in a query to minimize the required number of MR cycles. However, unbound-property queries would require processing a union of all VP property relations. The HadoopDB-based extension [14] uses a hybrid database-Hadoop architecture that exploits the partitioning scheme to push part of the execution into the database/RDF-3X. Hash partitioning on Subject can enable local evaluation of unbound-property star subpatterns. However, once the execution is handed over to Hadoop the redundancy in intermediate results impacts the disk I/O, sorting, and communication costs for the rest of the execution workflow. In order to minimize the data shuffle costs, MRShare [24] enables sharing of map output data across grouping operations on a common input relation. Some other works proposed a value-partitioning scheme [21] to manage reducer-unfriendly groups during the cube computation process, and a reducer-routing strategy [38] that groups intermediate keys to balance the data across reducers. The evaluation strategies proposed in this paper, i.e., lazy β-unnesting strategies, are in similar spirit.

Optimizing unbound-property queries: Earlier studies [35, 34] have shown that the vertical-partitioning (VP) [4] storage model may be inefficient for unbound-property queries. Such queries result in multiple joins and large unions of VP relations, which gets worse for data containing large number of property types. The multi-indexing schemes in systems such as RDF-3x [22] could benefit single-star unbound-property queries. However, such systems may not scale well for large RDF graphs, particularly for queries with low selectivity and unbound objects [15]. There have been efforts [36] to optimize simple unbound-property queries to RDF views over relational databases. Since naive translation of an unbound-property query into SQL results in unions of multiple subqueries, the proposed Group Common Term transformer [36] exploits common terms in complex disjunctive SQL queries and rewrites them into a smaller number of queries. Our work proposes a scalable solution for processing unbound-property queries on MapReduce-based parallel processing platforms.

Prior Work. A previous work explored the use of a non-relational data model and algebra, i.e., the *Nested TripleGroup Data Model and Algebra* (NTGA) [30, 17], for efficient RDF query processing on MapReduce. The NTGA allows an alternative interpretation of queries in terms of a "grouping" operation and a set of *triplegroups*, that enables shorter execution workflows when compared to relational query plans in systems such as Hive and Pig. For example, query $Q1$ requires 3 MR cycles altogether (two cycles for computing star-joins SJ_1, SJ_2, and a third cycle to join the stars) as shown in Figure 1, while the NTGA would compute both SJ_1 and SJ_2 in a single cycle using a "grouping" operation, followed by a second cycle to compute the join between the stars.

***Comparison with Redundancy due to Multi-valued Properties*.** Unlike the normalized representation of intermediate results of relational operations, the nested triplegroup data model can concisely represent intermediate results with multi-valued properties, e.g.,

{ ($gene9$, *xGO*, **{go1, go9}**) // A single triplegroup representing
($gene9$, *label*, retinoid...) // two n-tuples $t1$ and $t4$
($gene9$, *synonym*, RCoR-1)} // by nesting object component

Though the "nested object" model and *nesting-aware physical operators* [31, 29] reduce the I/O footprint of execution workflows, a join involving an unbound-property triple pattern would still produce 'n' triplegroups (assuming n triples with subject $gene9$). More importantly, all n triplegroups contain redundant bound-property component. In this paper, we generalize the concept of triplegroup nesting to allow *nesting of property-object components*, to implicitly represent intermediate results while evaluating unbound-property queries. However, such an implicit representation involves triples playing multiple roles, i.e., a triple may match the bound and the unbound component of a query, which needs to be incorporated into the "unnest" process, referred here after as β-`unnest`. Additionally, there are implications of when and what portion of a triplegroup is β-unnested during the different phases of an execution workflow, resulting in choices for evaluation strategies. Specifically, this paper makes the following contributions:

- We introduce new logical operators and query rewrite rules that allow the translation of unbound-property queries into NTGA-based logical plans. The correctness and sufficiency of query rewrite rules is also presented.
- We introduce new physical operators that offer different evaluation strategies - *eager vs. lazy β-unnesting* of intermediate results during query processing.
- Extensive evaluation using large RDF graphs, both Semantic Web synthetic benchmark and real-world biological datasets, demonstrates the efficiency of our approach over relational-style processing of unbound-property queries in Pig and Hive.

2. PRELIMINARIES

2.1 MapReduce and Data Processing

In the MapReduce programming model, data processing tasks are encoded as *map* and *reduce* functions, that are executed in parallel across a cluster of computing nodes. Relational operations such as a join between two relations, maps to a processing cycle consisting of two phases – the *Map* phase and the *Reduce* phase. In the *Map* phase, a set of slave nodes (*mappers*) execute the *map* function that tags each tuple based on the join key. Map output tuples are partitioned on the join key and *shuffled* across the network to another set of slave nodes (*reducers*). In the *Reduce* phase, each reducer receives a collection of tuples with the same join key, and computes the join. The output of the Reduce phase is written onto the *Hadoop Distributed File System* (HDFS) and read back in a subsequent cycle. Each MapReduce (MR) cycle involves costs associated with initial input data reads in the map phase (M_{Read}), the data shuffling costs between mappers and reducers that involve local disk writes at the mappers (M_{Write}), sort-merge costs (MR_{Sort}) as well as network transfer costs (MR_{TR}), and finally the cost of writing the reduce output to the HDFS (R_{Write}).

To evaluate graph pattern queries on MapReduce, one can exploit the fact that graph pattern queries often consist of multiple star-structured subqueries e.g., SJ_1 and SJ_2 rooted at variables *?s1* and *?s2* in query $Q1$, that can be evaluated using a multi-way join algorithm. For a graph pattern query with l star subpatterns, the typical MapReduce execution plan generated by relational-like platforms such as Hive and Pig consist of a sequence of MapReduce cycles $MR_1, MR_2, .., MR_n$ such that $1 \leq n \leq (l-1)$ cycles are used for executing the l star-joins, and $(n - l)$ MapReduce cycles for the remaining joins in the query. Our example query $Q1$ can be evaluated in 3 MR cycles as shown in Figure 1: MR_{SJ1} and MR_{SJ2} to compute star subpatterns SJ_1 and SJ_2 respectively, followed by a third cycle MR_{J1} to join the stars. Given such a MR workflow W, the overall processing cost of W is:

$$\text{Cost}(W) = \text{cost}(MR_1) + \text{cost}(MR_2) + ... + \text{cost}(MR_n)$$

where the I/O, sorting, and network transfer costs of each cycle compound across multiple cycles of a lengthy workflow. Furthermore, the portion of redundant data in the intermediate results directly impacts the HDFS writes (R_{Write}) for the current MR cycle, and the scan costs (M_{Read}) and shuffle costs (MR_{Sh}) of subsequent MR cycles. Hence, the redundancy has a ripple effect on the costs of reads, writes, sorting and the data transfer costs across a workflow with multiple MR cycles. Thus, lengthy workflows lead to performance inefficiency and an important optimization goal is to *minimize the length of an MR execution workflow* [6, 15, 40].

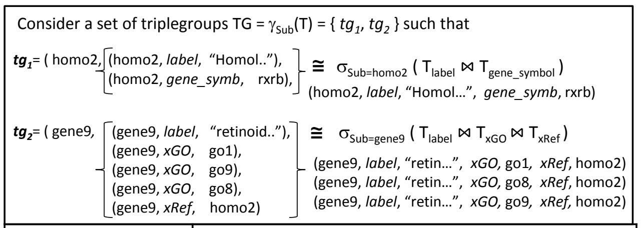
However, grouping of joins based on star structures does not necessarily result in the typical join order generated using traditional cost-based optimization. One challenge is that most cloud processing platforms are used in an on-demand model, where pre-computed statistics for cost-based optimization may not be available or take too long to compute, resulting in long lead times. More importantly, ordering joins in terms of their costs may generate some linear subplans requiring one input as the full triple relation, which in the absence of an index is a full scan. Such plans may incur larger overhead due to HDFS reads, which outweighs the savings achieved by pushing selective joins ahead.

Our previous work [30, 17] explored an algebraic optimization technique that rewrites graph pattern queries using operators that are more MapReduce-cognizant. It has been demonstrated that the underlying data model and algebra called the *Nested Triple-Group Data Model and Algebra* (NTGA), not only results in short execution workflows [30, 17], but also enable scan-sharing [18] across star subpatterns, while reducing the I/O footprint of intermediate results [31, 29]. In the next section, we overview the data model and algebraic operators in NTGA that enable nimble execution workflows while evaluating RDF graph pattern queries on MapReduce.

2.2 TripleGroup-based Processing of Graph Pattern Queries on MapReduce

The NTGA data model represents the RDF database as sets of related "group of triples" or *TripleGroups*. For example, triples in the database can be modeled as a set of *Subject TripleGroups*, each consisting of triples that share a common subject. For example, triplegroups tg_1 and tg_2 in Figure 2 represent subject triplegroups corresponding to triples sharing common subjects gene9 and homo2, respectively. Given such a data model, answering graph pattern queries translates to manipulation of triplegroups. Some of the most relevant triplegroup operators are summarized in Figure 2 and discussed below.

Algebraic Operators. Consider a query Q' with two star subpatterns St_1={$label$, $gene\_symb$} and St_2={$label$, xGO, $xRef$}. NTGA's grouping operator (γ) computes a set of subject triplegroups TG based on the subject column as shown in Figure 2. Given such a set of triplegroups TG, a match to a star subpattern is a selection operation (σ^{γ}) that extracts a subset of triplegroups that match the required join structure, i.e., a valid triplegroup must contain at least one triple corresponding to each of the property types



Consider a set of triplegroups TG = γ_{Sub}(T) = { tg_1, tg_2 } such that

tg_1 = (homo2, {(homo2, *label*, "Homol.."), (homo2, *gene_symb*, rxrb)}) $\cong$ $\sigma_{Sub=homo2}$ ($T_{label} \bowtie T_{gene\_symbol}$)
(homo2, *label*, "Homol...", *gene_symb*, rxrb)

tg_2 = (gene9, {(gene9, *label*, "retinoid.."), (gene9, *xGO*, go1), (gene9, *xGO*, go9), (gene9, *xGO*, go8), (gene9, *xRef*, homo2)}) $\cong$ $\sigma_{Sub=gene9}$ ($T_{label} \bowtie T_{xGO} \bowtie T_{xRef}$)
(gene9, *label*, "retin...", *xGO*, go1, *xRef*, homo2)
(gene9, *label*, "retin...", *xGO*, go8, *xRef*, homo2)
(gene9, *label*, "retin...", *xGO*, go9, *xRef*, homo2)

Notation	Semantics
TripleGroup Filter $\sigma^{\gamma}_{P_{bnd}}$(TG)	Enforces structural constraints in a star subpattern by matching the set of bound properties P_{bnd} and eliminating triplegroups in TG that violate the required join structure (structure-based validation) e.g., $\sigma^{\gamma}_{\{label,\ gene\_symb\}}$(TG) = $TG_{\{label,\ gene\_symb\}}$ ={ tg_1}
TripleGroup Join $\bowtie^{\gamma}$(tp1:TG_1, tp2: TG_2)	Joins triplegroups tg_1 εTG_1 and tg_2 εTG_2, based on the join conditions specified by triple patterns tp1 and tp2 respectively. e.g., $\bowtie^{\gamma}$(?s1 *label* ?label1 :$TG_{\{label,\ gene\_symb\}}$, ?s2 *xRef* ?s1 :$TG_{\{label,\ xGO,\ xRef\}}$) = { *ntg* } ntg = { (gene9, (*label*, "retinoid..."), (*xGO*, go1), (*xGO*, go8), (*xGO*, go9), (*xRef*, {homo2, (*label*, "Homolog...") (*gene_symb*, rxrb)}) }

Figure 2: Example NTGA Operators

in the star subpattern. Triplegroup tg_1 is a valid match for St_1 and is said to belong to the equivalence class $TG_{\{label,gene\_symb\}}$ that defines its join structure. Further, matching multiple star subpatterns translates to a disjunctive selection based on the set of properties in each star subpattern. For example, the two star subpatterns in Q' can be computed as follows:

$$\sigma^{\gamma}_{(\{label,gene\_symb\} \vee \{label,xGO,xRef\})}(TG)$$

Joins between star subpatterns can be computed using the join operator ($\bowtie^{\gamma}$) that is semantically equivalent to the relational join operator but is defined on triplegroups. The object-subject join between triplegroups tg_1 and tg_2 results in a nested triplegroup ntg whose root is the triplegroup tg_1 and child triplegroup is tg_2. Before proceeding, we review the notion of *content-equivalence* that enables lossless translation between relational algebra and NTGA plans.

Relational Algebra $\leftrightarrow$ NTGA Plans. Triplegroups are '*content-equivalent*' (represented as $\cong$) to the set of n-tuples computed using a set of relational-style joins. Let Stp be a star subpattern comprising of the set of bound properties $\{P_1, P_2, ..., P_k\}$, and T_{Stp} be the join result of vertically partitioned subset relations $T_{P_1}, T_{P_2}, ..., T_{P_k}$. Let $T_{Stp(s)}$ represent the subset of T_{Stp} with subject Sub = s.

$$T_{Stp(s)} = \sigma_{Sub=s}(T_{P_1} \bowtie T_{P_2} \bowtie ... \bowtie T_{P_k})$$

Each tuple in $T_{Stp(s)}$ is of 3k arity (each property in Stp is associated with 3 columns). Let π_{P_i} denote the projection of the (Sub, Prop, Obj) columns corresponding to the parent relation T_{P_i} with bound-property P_i. Let tg_s represent the set union of triples formed by the 3 columns, i.e.

$$tg_s = \pi_{P_1}(T_{Stp(s)}) \cup \pi_{P_2}(T_{Stp(s)}) \cup ... \cup \pi_{P_k}(T_{Stp(s)})$$

In summary, the tuples in $T_{Stp(s)}$ can be vertically partitioned into 'triples' whose union is equivalent to a subject triplegroup tg_s in the NTGA data model. For our example data in Figure 2,

$$tg_1 \cong \sigma_{Sub=homo2}(T_{label} \bowtie T_{gene\_symb})$$
$$tg_2 \cong \sigma_{Sub=gene9}(T_{label} \bowtie T_{xGO} \bowtie T_{xRef})$$

Benefits of NTGA Query Plans. For a query with 'n' star subpatterns, NTGA can compute ALL star subpatterns concurrently using a single 'grouping' operation, by first 'grouping' the triples into subject triplegroups and then applying a disjunctive selection based on the multiple star subpatterns. This is in contrast to the relational-style approach where each star subpattern is evaluated as a relational-style join. The grouping-based star-join computation naturally fits the *map-group-reduce* theme in MapReduce, and translates to just one MR cycle for computing all star-joins in the query (as opposed to 'n' MR cycles using relational-style plans). In addition to the reduction in the number of required MR cycles, NTGA also results in reduced size of intermediate results. Multiple related n-tuples resulting from relational-style joins involving a multi-valued property are implicitly represented as a single triplegroup in NTGA. For example, the 3 n-tuples corresponding to Stp_2 containing a multi-valued property *xGO* are implicitly represented using a single triplegroup tg_2 as shown in Figure 2. This is specifically important in minimizing the I/O footprint of long MapReduce execution workflows while processing RDF graph pattern queries.

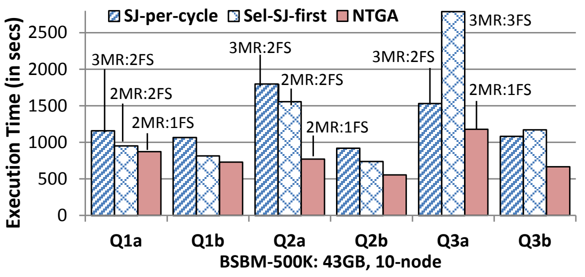
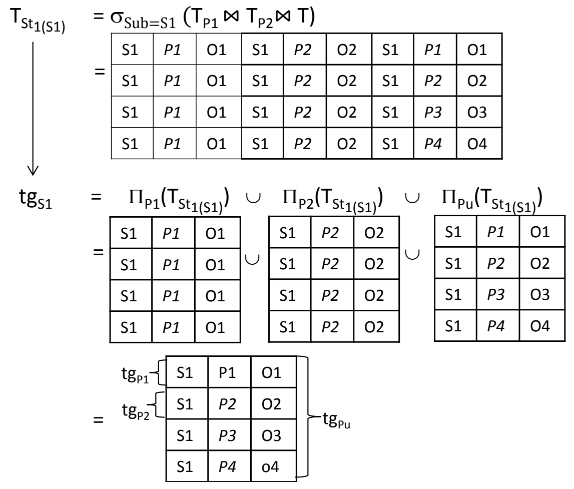


Figure 3: Evaluation of different groupings of star-joins (MR: No. of MapReduce cycles, FS: No. of Full Scans)

Consider a case study using 6 test queries (each with two star subpatterns) using the BSBM synthetic benchmark dataset (43GB) on a 10-node Hadoop cluster, as shown in Figure 3. The test queries have varying join structures with Object-Subject join (Q1a, Q1b, Q2a, Q2b) and Object-Object join (Q3a, Q3b) between star patterns. Queries Q1b, Q2b, Q3b are variations of Q1a, Q2a, Q3a respectively, where one of the two star-joins is highly selective due to an additional filter on the object column. Additional details about the evaluated queries are available on the project website [3]. We evaluated three different groupings of star subpatterns in a query, (i) a star-join per cycle approach (*SJ-per-cycle*), (ii) most selective grouping of joins first but preserving star structure as much as possible to minimize MR cycles (*Sel-SJ-first*), and (iii) concurrent evaluation of star-joins using the grouping-based approach in NTGA. *SJ-per-cycle* approach requires 3 MR cycles for all queries (2 of 3 cycles require full scan of triple relation). For Object-Subject joins, *Sel-SJ-first* approach can group joins into just 2 MR cycles (both cycles scan entire triple relation). For the Object-Object join (Q3a, Q3b), *Sel-SJ-first* still requires 3 MR cycles, but more importantly has very high HDFS reads due to full scan of triple relation in all 3 cycles. In contrast, the *NTGA* approach is able to minimize the number of MR cycles (2 cycles for all queries), as well as minimize the required number of full scans of the triple relation, thus outperforming the other two approaches for all test queries.

Earlier work on NTGA captures basic graph patterns. In this work, we build on the advantages of the TripleGroup data model and algebra for efficient evaluation of unbound-property graph pattern queries on MapReduce. Specifically, the semantics of the group-filter operator (σ^{γ}) requires all properties in the query structure to be bound. However, to capture more complex patterns, the algebra and the set of rewrite rules need to be extended. The following section introduces a number of extensions which allow us to relax the above constraint to provide an extended group-filter semantics for



$T_{St_1(S1)} = \sigma_{Sub=S1}(T_{P1} \bowtie T_{P2} \bowtie T)$

=

S1	*P1*	O1	S1	*P2*	O2	S1	*P1*	O1
S1	*P1*	O1	S1	*P2*	O2	S1	*P2*	O2
S1	*P1*	O1	S1	*P2*	O2	S1	*P3*	O3
S1	*P1*	O1	S1	*P2*	O2	S1	*P4*	O4

$tg_{S1} = \Pi_{P1}(T_{St_1(S1)}) \cup \Pi_{P2}(T_{St_1(S1)}) \cup \Pi_{Pu}(T_{St_1(S1)})$

=

S1	*P1*	O1
S1	*P1*	O1
S1	*P1*	O1
S1	*P1*	O1

$\cup$

S1	*P2*	O2
S1	*P2*	O2
S1	*P2*	O2
S1	*P2*	O2

$\cup$

S1	*P1*	O1
S1	*P2*	O2
S1	*P3*	O3
S1	*P4*	O4

=

tg_{P1}	S1	P1	O1	tg_{Pu}
tg_{P2}	S1	*P2*	O2	
	S1	*P3*	O3	
	S1	*P4*	o4	

Figure 4: Transformation: n-tuples to a triplegroup

evaluating unbound-property queries.

3. REWRITING UNBOUND-PROPERTY QUERIES USING NTGA

Consider an unbound-property star pattern $St_u = \{P_{bnd}, P_{unbnd}\}$ such that $P_{bnd} = \{P_1, P_2, ..., P_k\}$ represents the set of bound properties and P_{unbnd} represents an unbound property. Let T_{St_u} be the star-join result of relation $T(Sub, Prop, Obj)$ with vertically partitioned subset relations T_{P_1}, T_{P_2},...,T_{P_k}, and let $T_{St_u(s)}$ represent a subset of T_{St_u} with subject $Sub = s$.

$$T_{St_u(s)} = \sigma_{Sub=s}(T_{P_1} \bowtie T_{P_2} \bowtie ... \bowtie T_{P_k} \bowtie T)$$

The tuples in $T_{St_u(s)}$ have arity 3(k+1), where each property in St_u is associated with 3 columns in $T_{St_u(s)}$. Figure 4 represents the tuples in $T_{St_1(S1)}$ for a star-pattern St_1 with bound properties P_1, P_2 and an unbound property. To determine how St_u will be evaluated using NTGA, it will be useful to develop some correspondence between $T_{St_u(s)}$ and a subject triplegroup in NTGA. Note that a single triple may play multiple roles (occur multiple times) in the result of an unbound-property star pattern – one as a match for the bound property and the other as a match for the unbound property. For example, $(S1, P1, O1)$ in Figure 4 occurs once for the join with T_{P_1} and once for the join with T. In the NTGA data model, such multiple occurrences are implicitly represented once, which must be accounted for in the transformation process.

Continuing with the transformation process, let π_{P_i} and π_{P_u} denote the projection of the (Sub, Prop, Obj) columns corresponding to parent relation T_{P_i} with bound property P_i, and unbound property P_{unbnd} respectively. Let tg_s represent the set union of triples formed by the 3 columns, i.e.

$$tg_s = \pi_{P_1}(T_{St_u(s)}) \cup ... \cup \pi_{P_k}(T_{St_u(s)}) \cup \pi_{P_u}(T_{St_u(s)})$$

Figure 4 denotes the triples in tg_{S1} for our example star-pattern St_1, formed by the set union of the partitions π_{P_1}, π_{P_2}, and π_{P_u}. tg_s has the following properties:

(i) $\forall t_i, t_j \in tg_s$, the triples t_i, t_j agree on the subject column s.

(ii) $\exists$ non-empty subset of triples $tg_{P_i} \subseteq tg_s$ such that $tg_{P_i} = \pi_{P_i}(T_{St_u(s)})$, for each bound property $P_i \in P_{bnd}$.

(iii) $\exists$ a non-empty subset of triples $tg_{P_u} \subseteq tg_s$ such that $tg_{P_u} = \pi_{P_u}(T_{St_u(s)})$ and $tg_{P_u} \cap (tg_{P_1} \cup ... \cup tg_{P_k})$ may be non-empty.

Essentially, the tuples in T_{St_u} can be horizontally partitioned into sets of tuples with the same Subject column, and each element in the partition can be vertically partitioned into 'triples' whose union is equivalent to a subject triplegroup tg_s in the NTGA data model. The use of set union instead of bag union ensures that we have a triplegroup. Further, subsets of triples in tg_s represent matches to the bound and unbound-property triple patterns in St_u. This process basically describes a sequence of translation steps from the relational algebra to NTGA. In other words,

$$\begin{aligned} T_{St_u(s)} &= \sigma_{Sub=s}(T_{P_1}) \bowtie ... \bowtie \sigma_{Sub=s}(T_{P_k}) \bowtie \sigma_{Sub=s}(T_{P_u}) \\ &= tg_{P_1} \bowtie ... \bowtie tg_{P_k} \bowtie tg_{P_u} \end{aligned}$$

Conversely, for our example star-pattern St_1 in Figure 4, tuples in $T_{St_1(S1)}$ are implicitly represented in tg_{S1} and can be produced by $(tg_{P_1} \bowtie tg_{P_2} \bowtie tg_{P_u})$. A useful property is to distribute the join with the unbound-property triple pattern across a union of subset relations of T. In other words, if the triple relation T can be partitioned into two subset relations, i.e., $T = \{T_{P'_u} \cup T_{P''_u}\}$. Then by the distributivity of join over union, we have:

$$T_{P_{bnd}} \bowtie (T_{P'_u} \cup T_{P''_u}) \equiv (T_{P_{bnd}} \bowtie T_{P'_u}) \cup (T_{P_{bnd}} \bowtie T_{P''_u})$$

Evaluating St_u using NTGA requires applying group filter (σ^γ) to match the required query structures. Recall that σ^γ is defined in terms of a set of bound properties. One might consider evaluating an unbound-property star-pattern query using σ^γ with a disjunction of concrete pattern combinations. Each such combination will consist of the set of bound properties P_{bnd} with each property in the database. For example, if $P_{bnd} = \{P_1, P_2\}$ is the set of bound-properties in the star pattern and $P = \{P_1, P_2, ..., P_{10}\}$ represents the set of all properties in the database. Then, the σ^γ expression is:

$$\sigma^\gamma_{(\{P_1,P_2,P_1\} \vee \{P_1,P_2,P_2\} \vee ... \vee \{P_1,P_2,P_{10}\})}(TG)$$

This would filter out triplegroups that do not match any of the required pattern combinations. However, the approach of enumerating all possible pattern combinations may be inefficient depending on the number of properties in the database. Additionally, the subject triplegroup tg_s may contain additional triples relevant to other patterns, and hence may not exactly match a single pattern combination. Hence, there is a need to relax the σ^γ to restrict the matching of structural constraints to the bound properties of the unbound-property star pattern. This means that triplegroups that contain all the bound properties (may contain additional properties), should be produced as part of the result for σ^γ. Once this is done, we need to extract subsets of triples in tg_s that are exact matches for any of the required pattern combinations. This is achieved by extracting the subset of triples corresponding to P_{bnd} and generating their union with each triple in the unbound-property subset tg_{P_u}. In the following section, we provide the formal definitions for a specialized group-filter operator ($\sigma^{\beta\gamma}$) and the unnest operator (β-`unnest`) that extracts the perfect matches to the unbound-property star-pattern. From here on, we assume the convenience function $tg.props()$ ($st.props()$) to retrieve the set of properties in a triplegroup tg (star pattern st).

DEFINITION 1. *(β **Group-filter**) Given a set of subject triplegroups TG and a star pattern $St_u = \{P_{bnd}, P_{unbnd}\}$ contaning an unbound property, the β **group-filter** operator $\sigma^{\beta\gamma}$ returns the subset of triplegroups in TG that contain a non-empty subset of triples matching all bound properties P_{bnd}. Specifically,*

$$\sigma^{\beta\gamma}_{(P_{bnd}, P_{unbnd})}(TG) := \{\ tg_i \in TG \mid P_{bnd} \subseteq tg_i.props()\}$$

Essentially, $\sigma^{\beta\gamma}$ ensures that triplegroups contain a matching triple for each of the bound properties in P_{bnd}. Additionally, triplegroups

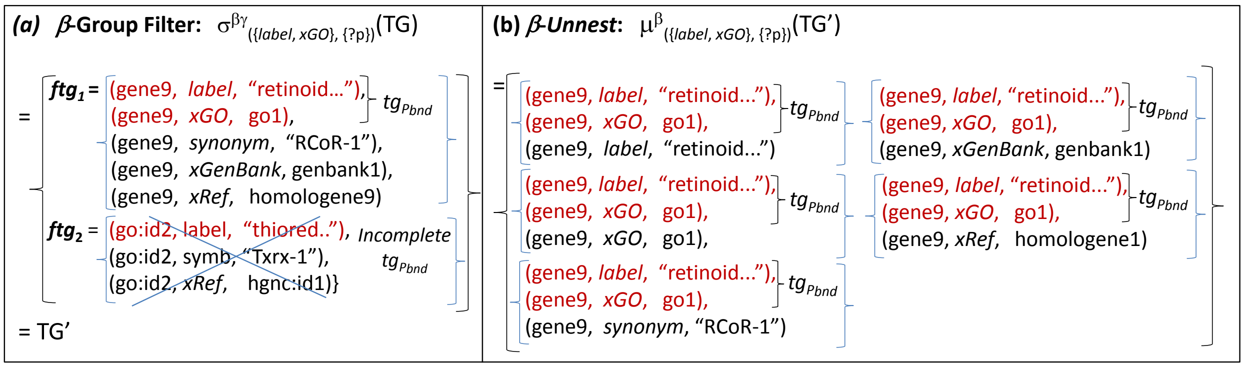

(a) β-Group Filter: σ^{βγ}({label, xGO}, {?p})(TG)
=
ftg1 = (gene9, label, "retinoid…"),
(gene9, xGO, go1),
tgPbnd
(gene9, synonym, "RCoR-1"),
(gene9, xGenBank, genbank1),
(gene9, xRef, homologene9)
ftg2 = (go:id2, label, "thiored.."),
Incomplete tgPbnd
(go:id2, symb, "Txrx-1"),
(go:id2, xRef, hgnc:id1)}
= TG'
(b) β-Unnest: μ^β({label, xGO}, {?p})(TG')
=
(gene9, label, "retinoid…"),
(gene9, xGO, go1),
(gene9, label, "retinoid…")
(gene9, xGenBank, genbank1)
(gene9, xGO, go1),
(gene9, xRef, homologene1)
(gene9, synonym, "RCoR-1")
tgPbnd

Figure 5: NTGA logical operators to evaluate unbound-property star-patterns

may also contain triples containing other property types. For example, given $P_{bnd} = \{label, xGO\}$, triplegroup ftg_1 forms a valid result for the $\sigma^{\beta\gamma}$ expression in Figure 5(a). However, ftg_2 does not contain a matching triple for the bound property xGO and hence gets filtered out.

Definition 2. *(β **unnest**) Given a set of triplegroups TG and an unbound-property star pattern $St_u = \{P_{bnd}, P_{unbnd}\}$, the **unnest** operator μ^β creates a set of triplegroups that are exact matches to St_u. Specifically,*

$$\mu^{\beta}_{(P_{bnd},P_{unbnd})}(TG) := \{\ tg_i = \{tg_{P_{bnd}} \cup t_i\} \mid tg_{P_{bnd}}, t_i \subseteq tg,\ tg_{P_{bnd}}.props() = P_{bnd},\ tg \in TG\ \}$$

In other words, the β-unnest operator extracts subsets of triples in a triplegroup tg that match the different pattern combinations corresponding to the unbound-property star-pattern. Figure 5(b) shows the 5 perfect triplegroups that are produced by β-unnesting the triplegroup ftg in Figure 5(a), each containing a subset of triples $tg_{P_{bnd}}$ matching the set of bound properties P_{bnd}, and a triple t_i that matches the unbound-property triple pattern.

Lemma 1. *Given a triple relation T and an unbound-property star pattern $St_u = \{P_{bnd}, P_{unbnd}\}$ such that the set of bound properties $P_{bnd} = \{P_1, P_2, ..., P_k\}$ and P_{unbnd} represents a single unbound property, the following equivalence holds:*

$$(T_{P_1} \bowtie ... \bowtie T_{P_k} \bowtie T) \cong \mu^{\beta}_{P_{bnd}}(\ \sigma^{\beta\gamma}_{(P_{bnd},P_{unbnd})}(\ \gamma_s(T)))$$

Proof: Let T_{St_u} and TG_{St_u} represent the set of tuples and triplegroups produced by evaluating an unbound-property star-pattern St_u using relational joins and NTGA respectively. We need to prove that all tuples in T_{St_u} are produced using NTGA. We prove by contradiction. Let us assume that there exists a tuple $tup_s \in T_{St_u}$ with subject s that cannot be produced using triples in tg_s. This can happen only if $\exists$ a triple $t_i \in tup_s$ such that $t_i \notin tg_s$. Firstly, since $t_i \in tup_s$, we know that the subject of t_i is s. If $t_i.\texttt{props}() \in P_{bnd}$, since t_i's subject is s, the $\sigma^{\beta\gamma}$ ensures that $t_i \in tg_s$. If $t_i.\texttt{props}() \notin P_{bnd}$, then $\sigma^{\beta\gamma}$ still retains t_i since its subject is s. Hence, $t_i \in tg_s$. The only other case is when a triple t_i plays multiples roles (matches both bound and unbound parts) which are implicitly represented in our data model. We rely on the correctness of the μ^β operator (illustrated earlier but proof omitted for brevity) to complete the proof.

Generalization to Multiple Unbound Properties. The β-unnest operator can be generalized to star-patterns containing multiple unbound-property triple patterns. Let P_α, P_β,...,P_m represent the m unbound properties in a star-pattern. Then the β-unnest operator results in a set of triplegroups $\{tg_{\alpha\beta...m}\}$ each containing the bound-property subset $tg_{P_{bnd}}$ and m triples, one each matching the unbound properties P_α, P_β,...,P_m.

$$\mu^{\beta}_{(P_{bnd},\{P_\alpha,P_\beta,..,P_m\})}(TG) := \{\ \{tg_{P_{bnd}} \cup t_\alpha \cup t_\beta \cup ... \cup t_m\}\ \}$$

such that $tg_{P_{bnd}} \subseteq tg$ is the bound-property subset, i.e., $tg_{P_{bnd}}.props()$ = P_{bnd}, and triples $t_\alpha, t_\beta, ...t_m \subseteq tg \in TG$.

4. TRANSLATION TO MAPREDUCE PLANS

The logical operators proposed in the previous section are integrated into RAPID+ [17] (an NTGA-based extension of Apache Pig). The query compilation process in RAPID+ begins with plans of logical operators, which are compiled to plans of physical operators, which could either be a single function or a function pair corresponding to the map and reduce phases of the logical operator. The MR plan is an assignment of physical operators to MR cycles.

The MR plan for an unbound-property query, executes the β-group-filtering using the `TG_UnbGrpFilter` ($\sigma^{\beta\gamma}$) operator in the reduce of the `TG_GroupBy`. This is followed by the β-`unnest` (μ^β) operator that produces a set of perfect triplegroups. Thus, both `TG_UnbGrpFilter` and `unnest` can be executed in the reduce of `TG_GroupBy` in a single MR cycle (MR_1). We call this as *eager* β-unnesting of triplegroups, represented in Figure 6(a). The joins between the triplegroups matching the different subpatterns can be computed using NTGA's `TG_Join` operator in the subsequent MR cycles. At the end of MR_1 for this strategy, we have intermediate results (perfect triplegroups for the star pattern subqueries) that contain redundancy with respect to the bound-properties. This increases the cost of $MR_1.R_{Write}$ and HDFS read ($MR_i.M_{Read}$) and shuffle costs ($MR_i.MR_{Shuffle}$) for subsequent cycles MR_i that process the output of MR_1. Therefore, optimization strategies to minimize the redundancy in intermediate results of the star-join computation phase would be useful to generate cost-effective MapReduce workflows.

4.1 Optimization using β-Unnesting Strategies

The intuition is to concisely represent the result of an unbound-property star-pattern as far along the MR workflow as possible. Unbound-property query structures such as $B4$ in Figure 8 do not involve further joins based on the bindings of the unbound-property triple pattern, and thus can remain in its (nested) implicit representation till the end of the MR workflow. Query structures such as our example query $Q1$ participate in joins based on the Object column of the unbound-property triple pattern. Hence, the star-join results for such star subpatterns need to be β-unnested before the join, since the map phase of `TG_Join` tags the triplegroups based on the join key and partitions them to different reducers. We propose evaluation strategies to delay the β-unnesting of triplegroups.

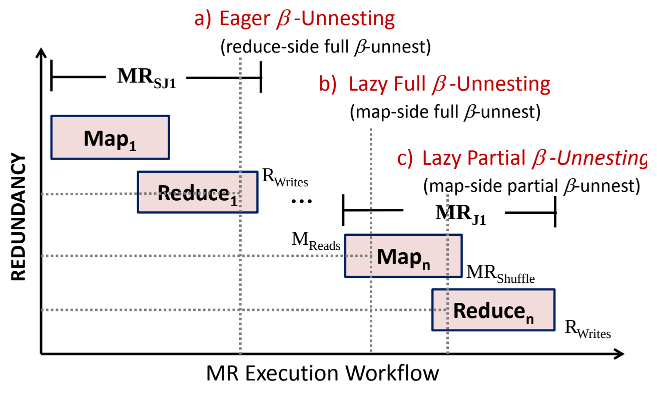


Figure 6: (a) eager β-unnest of a triplegroup during star-join, (b) lazy full and (c) lazy partial β-unnest in later join phase

Lazy Map-side β-Unnest: The β-unnesting of triplegroups can be delayed to a MR cycle that requires join on an unbound-property triple pattern, such as cycle MR_{J1} in Figure 6(b). Specifically, we push the β-`unnest` operator to the map phase of the corresponding `TG_Join` operator. We refer to the new physical operator as `TG_UnbJoin` (reduce phase remains same as `TG_Join`). By delaying the β-unnesting of triplegroups, we can minimize the redundancy in results of the star-join computation phase, and hence avoid unnecessary writes, reads, and shuffle costs for all subsequent intermediate MR phases. However, the β-`unnest` operator expands the map output of `TG_UnbJoin`, which impacts the shuffling costs. Assuming that `TG_UnbJoin` is assigned to the kth MR cycle MR_k in the workflow, then the redundancy in map output impacts $(MR_k.M_{Write} + MR_i.MR_{Sort} + MR_i.MR_{TR})$.

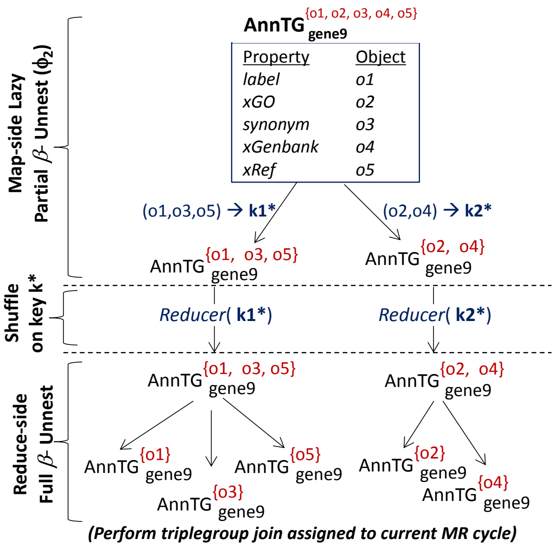


Figure 7: Lazy partial β-unnesting (ϕ_2)

Lazy Map-side Partial β-Unnest: We illustrate this strategy using Figure 7 . In order to support efficient look-up of (Property, Object) pairs in a triplegroup, we use an optimized internal representation scheme (extended multi-map) represented here as $AnnTG$, that concisely represents annotated triplegroups. Example annotated triplegroup $AnnTG^{\{o1,o2,o3,o4,o5\}}_{gene9}$ in Figure 7 represents the subject triplegroup ftg_1 (Figure 5(a)) which is a valid match for the unbound-property star subpattern SJ_2 in query $Q1$. Annotated TG $AnnTG^{\{o1,o2,o3,o4,o5\}}_{gene9}$ contains 2 bound-property triples (matching $label$ and xGO) and 5 triples matching the unbound-property triple pattern. A β-unnest operation produces 5 triplegroups (all containing the same bound-property component) that form a part of the map output for MR_k. The default partitioning scheme in Hadoop assigns the map output tuples to a reducer r based on the hash value of the join key, i.e., $hash(joinKey)\%r$. In the case that we have just 2 reducers, it is possible that triplegroups containing redundant bound-property component are partitioned and assigned to the same reducer based on the join keys (object of triples in the unbound-property component). For example, $AnnTG^{\{o1\}}_{gene9}$ and $AnnTG^{\{o3\}}_{gene9}$, may be assigned to the same Reducer, e.g., *Reducer1*. The redundancy in the map output of MR_k can be minimized if triplegroups that are eventually assigned to the same reducer are concisely represented during the shuffle phase, i.e., they are not β-unnested completely. By avoiding a part of the β-unnesting, we can reduce the size of map output, and hence reduce the shuffling costs. We propose a partial β-unnesting strategy that creates a set of triplegroups that each contain the bound-property component $tg_{P_{bnd}}$, and a subset of the unbound-property component $tg_{P_{unbnd}}$.

DEFINITION 3. *(**partial β-unnest**) Given a set of triplegroups TG, an unbound-property star-pattern St_u=$\{P_{bnd}, P_{unbnd}\}$, and a partition function ϕ_m that partitions the triples in $tg_{P_{unbnd}}$ into m partitions, the **partial-β-unnest** operator $\mu^{\beta'}$ produces a set of triplegroups such that:*

$$\mu^{\beta'}_{(P_{bnd},\phi_m)}(TG) := \{\ tg^i = \{tg_{P_{bnd}} \cup partition_i\}\ \}$$

where

- $\forall\ tg\ \in\ TG$, the bound-property subset $tg_{P_{bnd}}\ \subseteq\ tg$ such that $tg_{P_{bnd}}.props() = P_{bnd}$.
- A function ϕ_m assigns a triple $t_j \in tg_{P_u} \subseteq tg$ to $partition_i$, i.e., $\phi_m : t_j \rightarrow partition_i$, where $i \in \{1, 2, ..., m\}$.

The function ϕ partitions the triples in $tg_{P_{unbnd}}$ into m buckets based on the value of the join key. Essentially, $\mu^{\beta'}$ produces a maximum of m triplegroups for each triplegroup $tg \in TG$. For example, a partial β-unnest on $AnnTG^{\{o1,o2,o3,o4,o5\}}_{gene9}$ in Figure 7 using the partition function ϕ_2 produces 2 triplegroups - $AnnTG^{\{o1,o3,o5\}}_{gene9}$ and $AnnTG^{\{o2,o4\}}_{gene9}$ respectively. This implies that $\phi_2(o1) = \phi_2(o3) = \phi_2(o5) = k1^*$. Similarly, $AnnTG^{\{o2\}}_{gene9}$ and $AnnTG^{\{o4\}}_{gene9}$ are assigned to the same partition and hence remain implicitly represented as a single triplegroup. The redundant content in the map output is now a function of the partition range m. The partially β-unnested triplegroups are tagged and assigned to the reducers based on the partition key k^*. Triplegroup join with lazy partial β-unnest is implemented as a new physical operator, `TG_OptUnbJoin`. Figure 6(c) represents how the I/O footprint can be reduced by partial and delayed β-unnesting at map phase of MR_{J1}.

4.1.1 Algorithms For Physical Operators:

Algorithm 1 gives an overview of the job workflow for two key phases in the NTGA plan – Job_1, that computes 'matching' triplegroup equivalence classes that match all star subpatterns in the query, and Job_i, that computes the join between the triplegroup equivalence classes.

Job_1***: Compute 'matching' TG equivalence classes.*** The input to this job is a set of 3-tuples (triples) in the RDF database, and the output is a set of annotated triplegroups $AnnTG$ that match the star subpatterns in the query. In the map phase, each tuple is tagged based on the Subject component. In the reduce phase, all

```
Algorithm 1: MR job workflow for NTGA plan
  $Job_1$: Compute 'matching' triplegroup equivalence classes
    Map:
      $TG\_GroupBy$.Map(Tuples $T$);
    Reduce:
      $TG \leftarrow TG\_GroupBy$.Reduce($Sub$, List <Tuples>);
      $TG' \leftarrow TG\_UnbGrpFilter(TG, <EC, \{P_{bnd}, P_{unbnd}\}>)$;
  $Job_i$: Join between triplegroup equivalence classes
    Map:
      $TG\_OptUnbJoin$.Map($TG'$) //partial $\beta$-unnest
      or $TG\_UnbJoin$.Map($TG'$) //$\beta$-unnest
    Reduce:
      $TG'' \leftarrow TG\_OptUnbJoin$.Reduce($TG'$);
      or $TG'' \leftarrow TG\_UnbJoin$.Reduce($TG'$);
```

tuples corresponding to the same Subject component Sub are processed in the same reduce(), producing subject triplegroups. This is followed by a group-filtering phase to filter out triplegroups that violate the structural constraints in the query. Algorithm 2 shows the pseudocode for the β group-filtering operator, `TG_UnbGrpFilter`. The (Property, Object) pairs in a triplegroup (`tempMap` in line 1), are matched with all equivalence classes (star subpatterns) in the query (line 2). For each matching equivalence class EC, the bound properties P_{bnd} are extracted (line 4). The tuples in the group are considered relevant to the query only if they contain all bound properties (lines 5-9). If the matched equivalence class contains an unbound-property, the resultant $AnnTG$ contains all the (Property, Object) pairs for subject Sub (lines 6-7). If the matched equivalence class does not contain any unbound-property, only the relevant (Property, Object) pairs that match the bound properties are retrieved into the resultant triplegroup (line 8). Essentially, a group of tuples that does not contain the required set of bound properties for any of the star subpatterns in the query is filtered out.

```
Algorithm 2: TG_UnbGrpFilter
  $\beta$-GrpFilter ($tg$, $ECList$:<$EC$,{$P_{bnd}$, $P_{unbnd}$}>);
1 $tempMap \leftarrow$ extract triples in $tg$;
2 $matchedECList \leftarrow match(tempMap, ECList)$;
3 foreach $EC \in matchedECList$ do
4     $P_{bnd} \leftarrow$ extract bound properties in $EC$;
5     if $P_{bnd} \subseteq tempMap.keySet$ then
6         if $EC$ contains unbound property then
              //$\beta$ group filtering
7             $propMap \leftarrow tempMap$ ;
          else
              //Extract only bound properties in EC
8             $propMap \leftarrow$ extract $P_{bnd}$ entries from $tempMap$;
9         emit $\langle AnnTG(Sub, EC, propMap)\rangle$;
```

Job_i: ***Join between TG equivalence classes.*** The input to this phase is a set of annotated triplegroups, belonging to the two equivalence classes whose join is to be computed. The output is a set of annotated triplegroups, representing the joined result between the two equivalence classes. Based on the amount of redundancy in intermediate results due to the unbound-property star subpattern, a decision is made to either enable a partial or full β-unnest of the map output. Star subpatterns where the unbound-property is associated with a (partially) bound object, are not likely to cause redundancy, and hence a full β-unnest is enabled (`TG_UnbJoin` operator). For all other cases, the `TG_OptUnbJoin` operator is used.

Algorithm 3 shows the map-reduce functions for the operator `TG_OptUnbJoin` that integrates lazy partial-β-unnest operation. In the map phase, the annotated triplegroups that join on Subject are tagged using the Subject's partition key k* computed using ϕ_m (lines 1-3). For joins on Object, the AnnTG is partially β-unnested

```
Algorithm 3: TG_OptUnbJoin
   Map ($key$:null, $val$: AnnTG $atg$) ;
1  if join on $Sub$ then
2      $k$* $\leftarrow \phi_m(atg.Sub)$;
3      emit $\langle$ $k$*, $atg\rangle$ ;
   else if join on $Obj$ then
       //Partially $\beta$-unnest $atg$ using $\phi_m(Obj)$
4      $atgList \leftarrow$ partial-$\beta$-unnest ($atg$, $\phi_m$) ;
5      foreach $partialMap \in atgList$ do
6          $k$* $\leftarrow$ extract $k$* for $partialMap$ ;
7          emit $\langle$ $k$*, $partialMap\rangle$ ;

   Reduce ($key$:$k^*$, $val$:List of AnnTGs $TG'$) ;
8  $leftList \leftarrow \beta$-unnest leftEC AnnTGs from $TG'$;
9  $rightHash \leftarrow \beta$-unnest rightEC AnnTGs from $TG'$;
10 foreach $leftAnnTG \in leftList$ do
11     foreach $prop \in leftAnnTG.propMap$ do
           //Handle multi-valued property
12         $objList \leftarrow$ extract $prop$'s objects from $leftAnnTG$ ;
13         foreach $joinKey \in objList$ do
14             $rightAnnTG \leftarrow rightHash$.get($joinKey$) ;
15             emit $\langle$ joinTGs($leftAnnTG$, $rightAnnTG$)$\rangle$;
```

using the `partial-`β`-unnest` operation. The `partial-`β`-unnest` operator splits the (Property, Object) pairs in the triplegroup atg based on the Object's partition key resulting in a list of partially-unnested AnnTGs ($atgList$ in line 4). A map output tuple is generated for each partially-unnested AnnTG, tagged by its partition key k*(lines 5-7). The replication factor Rep is now a function of ϕ_m. In the reduce phase, all AnnTGs corresponding to the same group key k* but different join keys are processed in the same reduce(). In order to selectively join them based on the original join key, the AnnTGs corresponding to the right relation ($rightEC$) are β-unnested into perfect triplegroups and hashed based on the join key ($rightHash$ in line 9). The algorithm iterates through each AnnTG in the left relation ($leftEC$ in line 8), and probes the hashed relation ($rightHash$) based on the Object value (join key) for each property (lines 10-14). Multi-valued properties have multiple Object values and the probing is done for each value (lines 12-13). When a match is found, the two AnnTGs are joined (line 15) as per the definition of `TG_Join`. The partition factor used by ϕ depends on the size of input, potential redundancy factor, and average number of tuples that can be processed by a reducer.

5. EVALUATION

We evaluated the proposed algebraic optimization techniques on both real-world and synthetic datasets, and compared it with two popular relational-style MapReduce systems, Apache Pig and Hive. For NTGA, we evaluated two approaches for processing unbound-property graph pattern queries – *EagerUnnest* (Section 4), and the optimized *LazyUnnest* with map-side lazy β-unnesting. Experiments were conducted on NCSU's VCL [33], where each node in the cluster was a dual core Intel X86 machine with 2.33 GHz processor speed, 4G memory and running Red Hat Linux. 60 and 80-node Hadoop clusters (block size set to 256MB, 1GB heap-size for child jvms) were used with Pig release 0.11.1, Hive 0.10.0 and Hadoop 0.20.2. Only 20GB disk space was available per node, requiring large clusters to support large scale data, i.e., the 80-node Hadoop cluster made available ~1.6TB HDFS disk space. Results recorded were averaged over three trials.

Choice of Systems: Both Pig and Hive evaluate star-joins in a single MR cycle (one-star-join-per-cycle), resulting in same length workflows for all queries. Hive enables shared-scan of input relations within an MR cycle, thus minimizing the overall HDFS

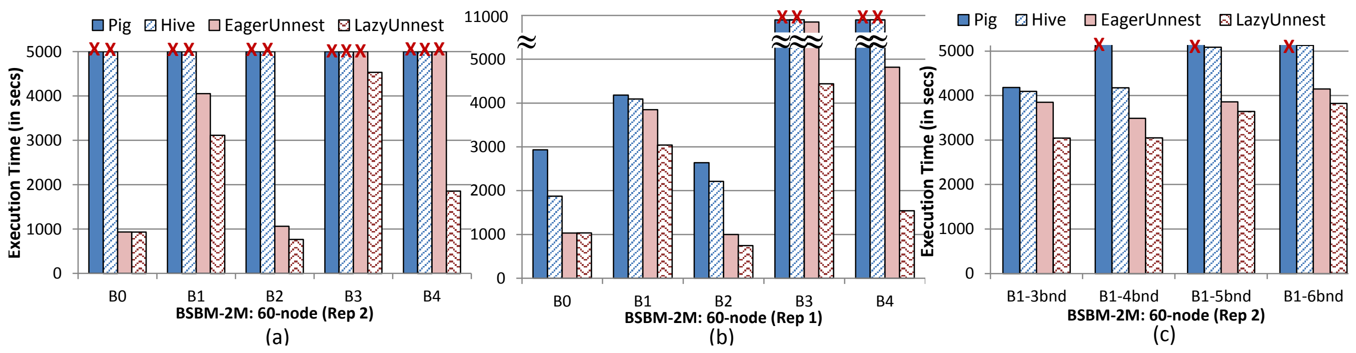


Figure 9: Performance with varying unbound-property star patterns with (a) replication factor 2 (b) replication factor 1 (c) Performance with varying size of bound-property component (BSBM-2M, 172GB, 60-node)

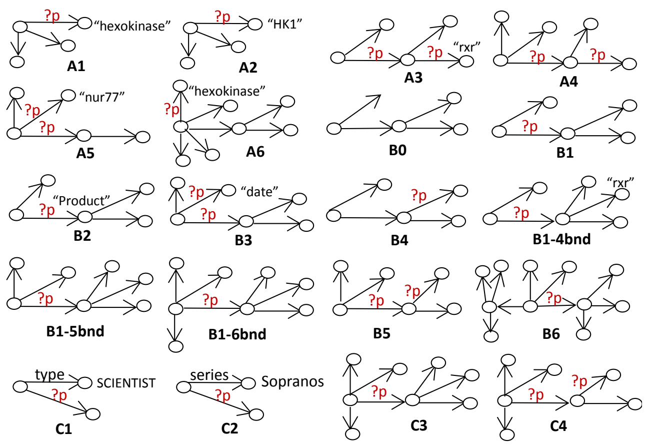


Figure 8: Testbed unbound-property RDF queries

reads. Pig can execute independent MR cycles concurrently, which is beneficial while evaluating multiple star subpatterns. NTGA approaches produce shorter workflows (all-star-joins in single MR cycle) when compared to Hive/Pig for queries with multiple star subpatterns. A triple relation is loaded as a 3-column table in Hive, where as Pig (and NTGA) process them as flat files. In Pig, the `SPLIT` operator is used to generate vertically-partitioning relations. HadoopRDF [15] does not currently support unbound-property queries and is not included for evaluation. Systems such as HadoopDB [14] scale well but rely on a heavy pre-processing phase that is more suitable for private clusters and less-evolving data. We focus on on-demand and pay-as-you-go workloads that involve quick exploration of datasets to get a sense of the data.

Testbed - Dataset and Queries: Real-world life sciences data from Bio2RDF [9] was used for evaluation. The queried biological data warehouse integrated 24 datasets, consisting of a total of ~4.7 billion triples (615GB in n-triple format). Two other real-world datasets, DBPedia Infobox (DbInfobox) [7] dataset of size 4.4GB (33.74M triples: 20.5M properties, 13.23M types) and the Billion Triple Challenge 2009 dataset (BTC-09) [1] of size 193GB (1.5B triples), were also used for evaluation. More than 45% of properties in both datasets are multi-valued with varying multiplicity. Two synthetic datasets generated by the BSBM [11] data generator tool – BSBM-1M (85GB dataset with 1 million Products, total ~370 million triples) and BSBM-2M (172GB dataset with 2 million Products, total ~700 million triples) were used for scalability study. The evaluation tested unbound-property queries with varying selectivity, varying join structures (single join to more complex structures with multiple star subpatterns) that are represented in Figure 8. Graph patterns in queries `A1`-`A6` have been extracted from Bio2RDF demo queries [2]. Additional details about the evaluated queries, along with the Pig / Hive scripts, are available on the project website [3].

Varying join structures (`B1`-`B6`): Scalability experiments were conducted to evaluate different join structures with varying number of unbound-property triple patterns, and varying arity of star subgraphs. Figures 9(a) and (b) show a performance comparison of Pig, Hive, and the NTGA approaches for two-star queries with no unbound properties (`B0`), one unbound-property triple pattern with join on unbound object (`B1`), one unbound property associated with a partially-bound object (`B2`), two unbound-property triple patterns in the same star with one partially-bound object (`B3`), and an unbound-property triple pattern (`B4`). Pig / Hive evaluate all three queries using 3 MR jobs (one per star-join), while NTGA evaluates them in 2 MR jobs. The queries involve a multi-valued property *prodFeature* that impacts redundancy.

In order to avoid data loss during node failure, fault-tolerant systems such as Hadoop rely on replication of data blocks on multiple nodes using a configurable parameter (dfs.replication). Initial set of experiments were conducted using a replication factor of 2 for the larger dataset BSBM-2M on a 60-node cluster (1.6TB disk space, 20GB per node). The results, shown in Figure 9(a), demonstrate how critical it is to concisely represent intermediate results and eliminate redundancy when possible. Missing bars marked with 'X' represent failed execution. Pig / Hive approaches failed during the last job (join between stars) for all 5 queries due to shortage of disk space. While *EagerUnnest* successfully executed for `B0`, `B1`, and `B2` by concisely representing subgraphs involving multi-valued properties, it failed for queries `B3` and `B4`. This is because the double unbound-property triple patterns in `B3` result in materialization of large intermediate results during the star-join computation phase, and we see the benefit of pushing the β-unnesting to a later phase (*LazyUnnest*) in executing this query. Similarly, for query `B4`, *LazyUnnest* successfully executes by materializing concise intermediate results, while other approaches fail.

In order to analyze the performance of the different approaches on the larger dataset, the same set of queries were repeated after reducing the HDFS replication factor to 1. Figure 9(b) shows the results comparing the performance of the approaches for BSBM-2M on the same 60-node cluster. In general, we see the benefit of the NTGA approaches for all queries. Query `B0` shows a baseline case with all bound properties where Hive and NTGA approaches outperform Pig due to scan-sharing. Further, NTGA approaches concisely represent results containing multi-valued property which leads to I/O savings. For query `B1` (join on unbound-property triple pattern), lazy partial β-unnesting reduces the shuffle

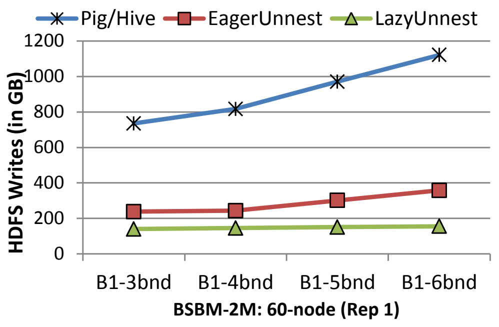


Figure 10: Total HDFS writes with varying size of bound-property component (BSBM-2M, 60-node)

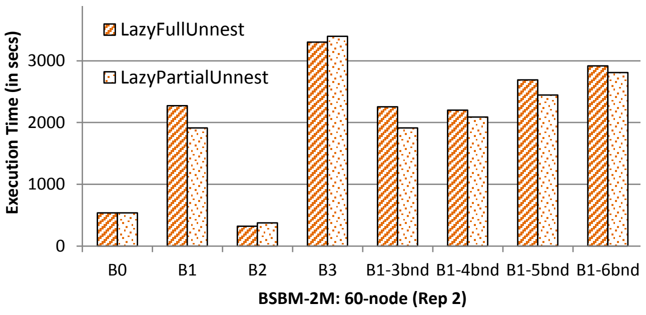


Figure 11: Lazy Full vs. Lazy Partial Unnesting: A comparative study of savings and overhead in MR cycle MR_{J1}

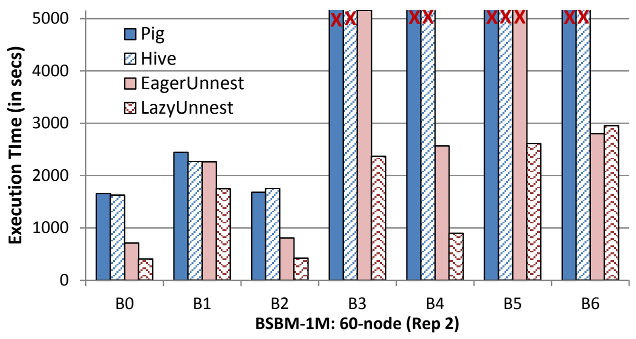


Figure 12: Performance comparison (BSBM-1M, 85GB)

costs and is 21% faster than eager β-unnesting (27% faster than Pig and 26% faster than Hive). For query B2, all approaches evaluate the filter on the partially-bound object associated with the unbound-property triple pattern in the initial map phase, and from there on, the execution is similar to the baseline query B0. As in the case of replication factor 2, Hive and Pig failed again for B3 and B4. The star subpattern with double unbound-property triple patterns (one with partially-bound object) in B3, is concisely represented in *LazyUnnest* with 80% less HDFS writes than *EagerUnnest*. In queries, such as B4, where the unbound-property triple pattern does not participate in join between stars, the lazy β-unnesting strategy keeps the result compact till the end, thus saving on intermediate disk reads / writes as well as final writes. Lazy β-unnesting using *LazyUnnest* results in 61% less HDFS writes than *EagerUnnest*, and overall has a 68% gain in performance times over the eager β-unnesting approach.

Choice of Lazy β-Unnesting Strategies: Testbed queries consist of varying structure of unbound-property triple patterns. For example, unbound-property triple patterns in queries B2 and B3 have partially-bound objects, i.e., the user does not know the exact property relationship but knows something about the object. In such cases, it is likely that the number of triples matching the unbound-property triple pattern are reduced and hence the associated star-join is more selective, i.e., results in less number of pattern combinations when compared to same triple pattern with an unbound object. Other queries such as B1 consist of unbound-property triple pattern with an unbound object. Though lazy β-unnesting is beneficial for all cases, we wanted to study benefits and overhead of lazy full and lazy partial β-unnest strategies. Figure 11 shows execution times for the last MR cycle (MR_{J1}) where the join involving the unbound-property triple pattern is computed. Since the size of input for MR_{J1} is same for both approaches, this analysis allows us to zoom into the map-side overhead for full and partial β-unnest, savings in shuffle costs, and analysis of reduce-side overhead in the case of partial-β-unnest. Our experiments show that a lazy full β-unnest may be sufficient for unbound-property queries with partially bound objects (queries B2 and B3). However, unbound-property queries with an unbound object ($B1$ series), benefit from partial-β-unnest. Other experiments were corroborative to these findings, and hence the *LazyUnnest* approach reported in this section evaluate lazy full-β-unnest for unbound-property queries with partially-bound-object patterns, and lazy partial-β-unnest for those with unbound-object patterns.

Varying number of bound-property edges: Unbound-property queries with bound-property triple patterns varying from 3 (B1-3bnd) to 6 (B1-6bnd) were evaluated. Figure 10 shows the total amount of HDFS writes for Pig, Hive and the NTGA approaches for the test queries evaluated on a 60-node cluster with BSBM-2M. In general, the increase in the number of bound-property components results in a gradual increase in the size of reduce output for Pig and Hive, while lazy β-unnesting keeps the result concise till the end of map phase of the last MR job (Job_2). The relational approaches produce 10 combinations of the bound component for the test queries since the relational arity of the subgraph that matches the unbound-property subpattern is 10. However, *LazyUnnest* compactly captures all the required combinations, resulting in approx. 80 to 86% less HDFS writes than Hive / Pig for queries B1-3bnd to B1-6bnd, respectively. Additionally, the reduce output for the NTGA approaches remain almost constant for such query patterns, which allows more flexible exploration of large datasets. Figure 9(c) shows a comparion of the execution times for all approaches. Note that Pig failed for all queries beyond three bound-property subpatterns. *LazyUnnest* (ϕ_{1K}) consistently outperformed the other approaches, running about 25% faster than Hive.

Varying size of RDF graphs: Figure 12 shows the evaluation of the BSBM queries using BSBM-1M (85GB) on the 60-node cluster (HDFS replication factor 2). NTGA approaches successfully executed for all datasets, with up to 80% less HDFS writes after the star-join computation phase for query B1 when compared to Hive. Once again it was observed that both Pig and Hive failed for queries B3 and B4 due to insufficient disk space. This is due to the high redundancy in star-join result that ripples into the next MR job, impacting the scan and I/O costs. For query B2, *LazyUnnest* outperforms all other approaches, executing about 75% faster than both Pig and Hive. *LazyUnnest* reduces the redundancy in intermediate results, and thus improves the execution time of the eager β-unnesting approach (*EagerUnnest*) by 54% (65%) for query B3 (B4). Hive / Pig failed to execute for more complex queries such as B5 and B6. These sets of experiments demonstrate the benefit of the proposed strategies in mitigating the effect of redundancy on MapReduce processing costs.

Real-world Unbound-property Queries (A1-A6): Figure 13

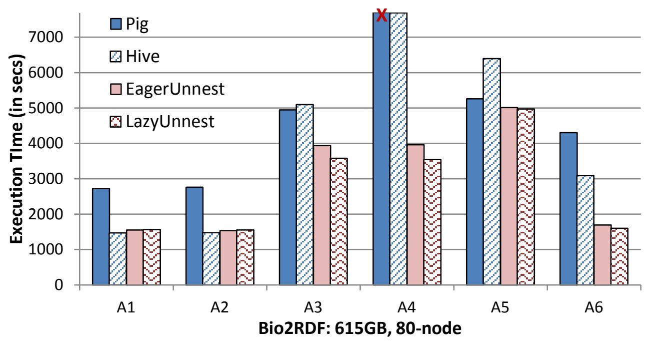


Figure 13: Evaluation of real-world unbound-property queries (Bio2RDF Life Sciences Dataset)

shows a performance comparison of Pig, Hive, and the two NTGA approaches for Bio2RDF queries `A1`-`A6` on a 80-node Hadoop cluster. Queries `A1` and `A2` have one star subpattern with one unbound-property triple pattern associated with partially-bound objects. For query `A1`, while Hive / Pig approaches produce all combinations of subtuples matching the bound property with triples matching the unbound property (∼63K tuples), *EagerUnnest* produces ∼7K triplegroups that concisely represent subtuples with multi-valued properties. *LazyUnnest* achieves more concise representation of all combinations corresponding to the unbound-property star pattern and produces only ∼3K triplegroups. The impact of the savings in HDFS writes due to elimination of redundancy in intermediate results, becomes more clear with the two-star queries (`A3`-`A6`).

Queries `A3` and `A4` contain an unbound-property in each of the two star subpatterns (one with partially-bound object). While Pig / Hive materialize 26GB of intermediate results in the star-join computation phase for query `A3`, the NTGA approaches write only about 1.3GB of data to the HDFS, contributing to the 32% performance gain over Hive while computing the star subpatterns. The *LazyUnnest* results in reduced HDFS writes in MR_1, and reduced scan costs and shuffling costs in MR_2, resulting in additional 18% performance gain over *EagerUnnest* in MR_2. For query `A4`, Pig initiates 4 MR jobs (initial map-only job to read entire input and compress it, 2nd and 3rd MR jobs to compute the two star patterns, and the 4th job to join the stars). However, Pig approach failed (marked as 'X') due to lack of HDFS space while executing the last job. Again, there is a huge savings in terms of HDFS writes, with *EagerUnnest* and *LazyUnnest* producing only 1.8GB and 0.6GB of intermediate results, respectively, after the initial star-join phase, as opposed to 152GB of writes in Hive. An important factor that results in large intermediate results with relational-style processing, is the redundancy due to the presence of large number of high multiplicity properties in biological datasets (representative of real-world datasets). For `A4`, *EagerUnnest* and *LazyUnnest* approaches are 48% and 53% faster than Hive, respectively.

Query `A5` contains a star pattern with two unbound-property triple patterns – one whose object matches a gene "nurr77", and the other with an unbound object, connecting the star to a single edge retrieving the *label* property type. Hive executes `A5` using 2 MR jobs, with both jobs requiring a full-table scan. NTGA approaches also execute using 2 MR jobs but with one full-table scan, resulting in overall savings of about 1400s (22% gain) over Hive. The single unbound-property triple pattern in query `A6` partially binds the object to "hexokinase". While Hive uses 3 MR jobs, including 2 for the star-join computation, Pig uses an extra map-only job to compress the input (total 4 jobs). NTGA's *LazyUnnest* approach shows a benefit of up to 48% over Hive.

DBPedia Queries (`C1`-`C4`): Additional experiments were conducted on varying sizes of real-world datasets, 4.3GB DBInfobox dataset (5-node cluster) and 193GB BTC-09 dataset (40-node cluster) as shown in Figure 14. Four different query structures were used. `C1` and `C2` are simple queries with single join that retrieve all information about Scientists (unselective) and Sopranos TV series (selective). In the case of DBInfobox dataset, since the data processed is quite small, the benefit of the NTGA approach is not seen for the first two queries. However, Pig does better than Hive since it processes two copies of the input relation, and hence initiates double the number of mappers and reducers. `C3` and `C4` represent real-world scenarios during exploration where the relationship between entities (star subpatterns) is unknown. NTGA approaches showed a performance gain of 20-22% and 50% over Hive and Pig respectively for query `C3`, and resulted in approx. 80% less HDFS writes than Hive. All four queries had redundancy factor greater than 0.6. In particular, `C4` which involved an unbound-property in each of the two star patterns showed a redundancy factor close to 0.89, and hence showed major improvement (50% gain over both Pig and Hive) with the lazy β-unnesting strategy.

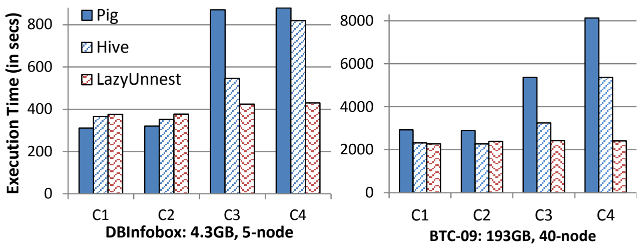


Figure 14: Evaluation of real-world unbound-property queries (DBInfobox and Billion Triple Challenge'09)

Unbound-property queries on the BTC-09 dataset resulted in very large HDFS reads which negatively impacted Pig the most, due to its multiple scans per star-join. The scan-sharing across star patterns in NTGA resulted in 50% less HDFS reads for the two star queries. NTGA approaches resulted in 54% (25%) gains over Pig (Hive) for query `C3` with 1 unbound-property. The result of the star-join phase for `C4` (2 unbound properties) has redundancy factor of 0.93 (0.75GB) and increases to 0.98 (14GB) in the final output for Pig/Hive. The lazy β-unnesting strategy results in 98% less HDFS writes, and have 70% (55%) performance gain over Pig (Hive) for `C4`. In general, real-world data contained multiple multi-valued properties with varying multiplicity, and highly benefited by the generalized nested representation of triplegroups and lazy β-unnesting strategies while processing unbound-property queries.

6. CONCLUSION

We propose a scalable solution for processing unbound-property graph pattern queries on MapReduce, by minimizing the redundancy in intermediate results that adds avoidable costs while processing long execution workflows. The proposed approach uses a nested triplegroup model to implicitly represent the intermediate results and lazily 'unnest' them only when necessary. A combination of the two result in significant savings in intermediate HDFS reads and writes, which form a major portion of query processing costs on MapReduce. Additional savings in intermediate map-reduce data shuffling costs can be achieved by delaying a portion of the 'unnest' to the reduce phase. Experiments show promising results for different query join structures with varying selectivities. Future directions include exploring more complex structures with multiple unbound-property patterns as well as unbound-property queries with aggregation constraints.

7. REFERENCES

[1] Billion triple challenge. *http://challenge.semanticweb.org/*.

[2] Bio2RDF Demo Queries. `http://sourceforge.net/apps/mediawiki/bio2rdf/index.php?title=Demo_queries`.

[3] Scaling Unbound-Property Queries using MapReduce. `http://research.csc.ncsu.edu/coul/RAPID/UnboundPropQueries`.

[4] D. Abadi, A. Marcus, S. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, 2007.

[5] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *VLDB*, 2009.

[6] F. Afrati and J. Ullman. Optimizing joins in a map-reduce environment. In *EDBT*, 2010.

[7] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives. Dbpedia: A nucleus for a web of open data. *ISWC/ASWC*, 2007.

[8] A. Bairoch, L. Bougueleret, S. Altairac, V. Amendolia, A. Auchincloss, G. A. Puy, K. Axelsen, D. Baratin, M.-C. Blatter, B. Boeckmann, et al. The universal protein resource (uniprot). *Nucleic Acids Research*, 36:D190–D195, 2008.

[9] F. Belleau, M.-A. Nolin, N. Tourigny, P. Rigault, and J. Morissette. Bio2rdf: towards a mashup to build bioinformatics knowledge systems. *Journal of biomedical informatics*, 41(5):706–716, 2008.

[10] A. Bialecki, M. Cafarella, D. Cutting, and O. O'MALLEY. Hadoop: a framework for running applications on large clusters built of commodity hardware. *Wiki at http://lucene.apache.org/hadoop*, 11, 2005.

[11] C. Bizer and A. Schultz. The berlin sparql benchmark. *IJSWIS*, 2009.

[12] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Comm. ACM*, 2008.

[13] I. Elghandour and A. Aboulnaga. Restore: Reusing results of mapreduce jobs. *VLDB*, 2012.

[14] J. Huang, D. Abadi, and K. Ren. Scalable sparql querying of large rdf graphs. *VLDB*, 4(11), 2011.

[15] M. Husain, J. McGlothlin, M. Masud, L. Khan, and B. Thuraisingham. Heuristics-based query processing for large rdf graphs using cloud computing. *TKDE*, 23(9), 2011.

[16] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *EuroSys*, 2007.

[17] H. Kim, P. Ravindra, and K. Anyanwu. From sparql to mapreduce: The journey using a nested triplegroup algebra. *VLDB*, 4(12), 2011.

[18] H. Kim, P. Ravindra, and K. Anyanwu. Scan-sharing for optimizing rdf graph pattern matching on mapreduce. In *CLOUD*, 2012.

[19] S. Kotoulas, J. Urbani, P. Boncz, and P. Mika. Robust runtime optimization and skew-resistant execution of analytical sparql queries on pig. In *The Semantic Web–ISWC*. 2012.

[20] H. Lim, H. Herodotou, and S. Babu. Stubby: a transformation-based optimizer for mapreduce workflows. *VLDB*, 2012.

[21] A. Nandi, C. Yu, P. Bohannon, and R. Ramakrishnan. Distributed cube materialization on holistic measures. In *ICDE*, 2011.

[22] T. Neumann and G. Weikum. The rdf-3x engine for scalable management of rdf data. *VLDB*, 2010.

[23] M.-A. Nolin, P. Ansell, F. Belleau, K. Idehen, P. Rigault, N. Tourigny, P. Roe, J. M. Hogan, and M. Dumontier. Bio2rdf network of linked data. In *Semantic Web Challenge; ISWC*, 2008.

[24] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. Mrshare: Sharing across multiple queries in mapreduce. *VLDB*, 2010.

[25] C. Olston, G. Chiou, L. Chitnis, F. Liu, Y. Han, M. Larsson, A. Neumann, V. Rao, V. Sankarasubramanian, S. Seth, et al. Nova: continuous pig/hadoop workflows. In *SIGMOD*, 2011.

[26] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, 2008.

[27] N. Papailiou, I. Konstantinou, D. Tsoumakos, and N. Koziris. H2rdf: adaptive query processing on rdf data in the cloud. In *WWW*. ACM, 2012.

[28] E. Prud'hommeaux and A. Seaborne. Sparql query language for rdf, W3C Recommendation, 2008. *URL http://www.w3.org/TR/rdf-sparql-query*, 2010.

[29] P. Ravindra and P. Anyanwu. Nesting strategies for enabling nimble mapreduce dataflows for large rdf data. *Int. J. Semantic Web Inf. Syst.*, 10(1), 2014.

[30] P. Ravindra, H. Kim, and K. Anyanwu. An intermediate algebra for optimizing rdf graph pattern matching on mapreduce. *The Semantic Web: Research and Applications*, 2011.

[31] P. Ravindra, H. Kim, and K. Anyanwu. To nest or not to nest, when and how much: representing intermediate results of graph pattern queries in mapreduce based processing. In *SWIM*, 2012.

[32] K. Rohloff and R. Schantz. High-performance, massively scalable distributed systems using the mapreduce software framework: The shard triple-store. In *PSI EtA*, page 4, 2010.

[33] H. Schaffer, S. Averitt, M. Hoit, A. Peeler, E. Sills, and M. Vouk. Ncsu's virtual computing lab: a cloud computing solution. *Computer*, 42(7), 2009.

[34] M. Schmidt, T. Hornung, N. Küchlin, G. Lausen, and C. Pinkel. An experimental comparison of rdf data management approaches in a sparql benchmark scenario. *ISWC*, 2008.

[35] L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, and S. Manegold. Column-store support for rdf data management: not all swans are white. *VLDB*, 2008.

[36] S. Stefanova and T. Risch. Optimizing unbound-property queries to rdf views of relational databases. In *SSWS*, 2011.

[37] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *VLDB*, 2(2), 2009.

[38] R. Vernica, M. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *SIGMOD*, 2010.

[39] X. Wang, C. Olston, A. Sarma, and R. Burns. Coscan: cooperative scan sharing in the cloud. In *SOCC*, 2011.

[40] S. Wu, F. Li, S. Mehrotra, and B. Ooi. Query optimization for massively parallel data processing. In *SOCC*, 2011.

Reaching a desired set of users via different paths: an online advertising technique on a micro-blogging platform

Milad Eftekhar
Department of Computer Science
University of Toronto
Toronto, ON, Canada
milad@cs.toronto.edu

Nick Koudas
Department of Computer Science
University of Toronto
Toronto, ON, Canada
koudas@cs.toronto.edu

Yashar Ganjali
Department of Computer Science
University of Toronto
yganjali@cs.toronto.edu

ABSTRACT

Social media and micro-blogging platforms have been successful for communication and information exchange enjoying vast number of user participation. Given their millions of users, it is natural that there is a lot of interest for marketing and advertising on these platforms as attested by the introduced advertising platforms on Twitter and Facebook.

In this paper, inspired by micro-blogging advertising platforms, we introduce two problems to aid ad and marketing campaigns. The first problem identifies topics (called *analogous* topics) that have approximately the same audience in a micro-blogging platform as a given query topic. The main idea is that by bidding on an analogous topic instead of the original query topic, we reach approximately the same audience while spending less of our budget. Then, we present algorithms to identify expert users on a given query topic and categorize these experts to finely understand their diversified expertise. This is imperative for word of mouth marketing where individuals have to be targeted precisely.

We evaluate our algorithms and solutions for both problems on a large dataset from Twitter attesting to their efficiency and accuracy compared with alternate approaches.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database applications—*Data mining*; J.4 [**Social and Behavioral Sciences**]: Economics

Keywords

Social media, Micro-blogging advertising platforms, Analogous topics, Alternative topics, Expert categorization

1. INTRODUCTION

Social media and micro-blogging have experienced exponential growth in user acquisition and participation over the last decade. Services such as Twitter, Facebook, and Pinterest allow millions of people to share billions of content and interact on a daily basis. Social platforms are targets of sophisticated advertising and marketing, mainly because of the large number of users, and the enormous amount of time users spend on them.

In micro-blogging platforms (e.g. Twitter), social connections get established by "following" an individual u. By establishing such a connection, you get to receive and view all posts (tweets) produced by u. The set of all posts that are visible by a user v is commonly referred to as the feed (timeline) of v. The act of following someone explicitly expresses interest in the information that person produces.

Social media and micro-blogging platforms are utilized by many as important marketing vehicles. By amassing a large number of followers, an individual or a company can broadcast messages targeted to these followers. Such messages vary depending on the type of the account (e.g., celebrity, professional, consulting, corporate) and what one wishes to achieve (e.g., brand/product awareness, sales leads, or general information dissemination). Typically, one produces information in the field of one's expertise – which is a topic or a set of topics that one knows well, professes, or is known for as an *expert* in the community. For example a celebrity (say a singer) will disseminate information of interest to fans, such as tour dates, personal events and announcements, as well as new songs and albums, whereas a company, say a technology startup, shares information related to its products, and the overall technology product space.

Recently, new advertising platforms have been introduced [10, 17]. In contrast to the keyword bidding model, as is popular in the case of search engine advertising, the micro-blogging platform takes a different approach. An advertiser selects a topic q, bids a specific dollar amount, and provides a post (known as a promoted post). The micro-blogging advertising platform identifies all the users that are *interested* in the topic q based on some internal algorithms and inserts the promoted post in the feed of these users (explicitly identifying it as a promoted post). The dollar amount is utilized by an auction that determines the winning bidder (for topic q). As an example, if we are interested in showing a promoted post to those users that are interested in *music*, we will bid an amount for the topic *music* and provide our promoted post. If we win, our promoted post will be inserted in the feed of those accounts interested in topic *music*. Commonly the amount we bid is per impression or per engagement (i.e., per person seeing or clicking on the promoted post).

In such a setting there are numerous opportunities for optimization. Of immediate interest would be reaching the same or approximately the same set of people with a lower cost. For example, by bidding on the topic *public relations* we can be successful only if we bid a price of x. What if we knew that if we bid on the topic *seo* (search engine optimization), we can reach the same people (and thus have the same impressions) for a price $y < x$? The first problem we outline in this paper is to produce a set of topics R *analogous* to a topic q (that we wish to bid on). These topics have the property that if we bid on one of them instead of q, our promoted post will be inserted in the feed of approximately (for a precise definition of approximate) the same people as those in the case of q. Now, by examining the associated cost of each topic, we can make a more informed decision by comparing the savings versus how many interested individuals our posts will reach for each of the analogous topics in R. We propose an algorithm called IAT to address this problem (Section 3).

Note that by advertising on a cheaper topic $t \in R$ instead of q, (1) (approximately) the same people see the ad and (2) expectedly same people engage with the ad. The cost we should spend in case of targeting t instead of q, therefore, would be lower (1) per impression (in cost per impression model) and (2) per engagement (in cost per engagement model). Hence by bidding on t, we reach same audience with a lower cost independent of the cost model in use.

Utilizing this technique provides a win-win situation for advertisers and the advertising platforms (e.g., Google, Twitter, etc.). Adopting the technique, advertisers who are interested in a topic will have more options (more topics with similar audience) to target. This prevents from the existence of a very popular topic that is too expensive to target alongside some cheaper topics that no one targets. In this situation, more advertisers afford to advertise. Hence the revenue of the advertising platform may significantly increase while advertisers also obtain more savings per advertisement.

A second popular marketing activity on micro-blogging platforms is to engage *experts* on specific topics into word of mouth marketing campaigns. By having experts on a topic become advocates of a product or a service, all of their followers become informed about the product or service. This is a typical form of word of mouth marketing. For example, if we are interested to market a new cloud computing product by word of mouth on a social media, we can engage cloud computing experts and persuade them to adopt, use, or talk about our product.

Finding the right advocates online is always challenging. Commonly, a user's account has a set of topics associated with it highlighting its expertise. Even if we have an a priori knowledge of the specific topics we wish our advocates to have expertise on, it may be impossible to find one that spans all these topics. Thus, a more iterative approach is desirable. Given that we can identify all experts on a single topic q, it would be very useful if we are capable of categorizing those experts based on other topics of their expertise. That would enable us to examine them in a more refined fashion and identify those that are closest to our topics of interest. For example, a set of experts in both *cloud computing* and *virtualization* may be more suitable for us than a set of experts in *cloud computing* and *data centers.* Being able to compute such expert groups algorithmically, given one specific topic of expertise q (cloud computing in our example), is imperative. We have typically no knowledge of what is the "right" number of groups and it is expected that some experts belong to many groups. We propose an algorithm (called CTE, Section 4) to group together all experts on any given topic in a varying number of groups (corresponding to high-level topics) based on the collective topics of expertise of all these users.

The problems discussed in this paper are inspired by social media and micro-blogging advertising platforms. Since the internal algorithms utilized by these platforms are unknown to public, we have proposed some models (e.g., expert identification, topic bidding model, etc. that are explained in the next sections) and utilized them in this paper as a proof of concept. We note that these models, our assumptions, and our methods are *not* based on or designed for any specific social media or micro-blogging platform.

As we have access to a large dataset from Twitter, we evaluate the algorithms on this dataset for various queries. Both IAT and CTE algorithms operate fast (a few minutes) in all experiments stressing the practicality of our developments. In addition, we deploy a qualitative study demonstrating the goodness of our findings and compare our CTE algorithm with some baseline techniques (Section 5). A literature review is provided in Section 6, followed by Section 7 concluding our discussion.

2. THE TARGETS

Different social media and micro-blogging platforms such as Twitter, Facebook, and Google+ have introduced the concept of *lists* (*circles* in case of Google+). A list is a user-defined set of accounts. Commonly, users create a list grouping their favourite accounts on a particular topic into that list which they annotate with a descriptive title. For example, in Twitter a user may create a list with the title of "politics" that include Twitter accounts @BarakObama, @AngelaMerkel, @HillaryClinton, @JohnKerry, and @DavidCameron. The utility of a list is to provide quick filtering (by list title) of posts from accounts belonging in the list. It is very typical to group together accounts that profess or depict expertise on a particular topic. A user can create multiple lists and an account can belong to any number of lists.

We utilize the infrastructure of the Peckalytics system [2] to associate with each account u, a set of topics T_u extracted from the titles of the lists containing that account. The process of extraction includes tokenization of the title, common word (stop word) and spam filtering, entity extraction, and related word grouping via Wikipedia and WordNet. The end result is, for each account u, a set of topics that best describes the topics associated (by other users) with u. We emphasize however, that *any* process of mapping an account to a set of topics that best describes the account can be utilized (e.g., machine learning methods). The techniques presented herein will work fine without any modification.

A user $u \in U$ is an *expert* on topic $t \in T$, iff $t \in T_u$. This means that (for our specific way of extracting topics) other users recognize u as an expert on topic t. We call topic t, a topic of expertise for u. The set of experts on topic t is denoted by E_t. A user $u \in U$ is *interested* in topic $t \in T$ iff the probability that u follows (reads) any content (a post, a shared video, a posted link, etc.) that is related to topic t is higher than a given threshold $\theta \in [0, 1]$. For a topic t, we refer to the set of all users who are interested in t as the *target set* of topic t denoted by S_t.

Micro-blogging platforms utilize several factors (content of posts, followers, etc.) to identify the interests of users and subsequently form target sets. However, such factors are largely proprietary. In this paper, we approximate the target set of a topic t by partitioning it into two categories: (1) users interested in t who are also expert on t (E_t) and (2) users interested in t who are not expert on t (I_t). In other words, users in E_t are producers of contents related to t, and users in I_t are consumers of contents related to t. Thus, $S_t = E_t \cup I_t$. For any topic t, the set of experts E_t is available to us; i.e., $E_t = \{u | t \in T_u\}$. However, the I_t sets are unknown to us (i.e., we do not know which users are interested in a given topic t). One may suggest to retrieve the interests for each user by taking the union of expertise topics of all accounts this user follows. This approach has some drawbacks. A given user (say u) may be an expert on several topics. When another user (say v) follows u, the user v may be interested in any of these topics but not necessarily in all of them. It is not straightforward to determine which topic is of interest to v, given the topics of u's expertise. In section 3.1, we present an approach to resolve this issue.

3. ANALOGOUS TOPICS

In an advertising scenario on a social media platform, by placing a bid for a particular topic q, assuming that the bid is granted, users in S_q will observe the promoted post on their feeds. Naturally, an interesting question is whether there is any other topic t that is cheaper than q (i.e., it is possible for a lower bid to be granted) with a target set S_t "close" to S_q. If possible, this would reduce advertising cost. This question is the key component of this Section. To formalize the question, we introduce some definitions.

Definition 1. *When a promoted post corresponding to a topic q is shown to a user belonging to $S_t \cap S_q$ (for some topic t), we say a* true impression *is achieved. If the promoted post is shown to a user in $S_t \backslash S_q$, we call that a* false impression. *Note that $X \backslash Y$ denotes the set difference between two arbitrary sets X and Y. As users in $S_t \backslash S_q$ are not interested in q, presenting a promoted post to them is not a desired outcome.*

Definition 2. *The* distance *between two arbitrary sets X and Y, denoted by $D(X,Y)$, is the size of the symmetric difference between them: $D(X,Y) = |(X\backslash Y) \cup (Y \backslash X)|$. Moreover, the* distance *between two topics q and t is the distance between their target sets S_q and S_t: i.e., $D(q,t) = D(S_q, S_t)$.*

We note that a low distance between topics q and t translates to a high true impression and a low false impression since $D(S_q, S_t) = |(S_t\backslash S_q) \cup (S_q \backslash S_t)| = |S_t \backslash S_q| + |S_q| - |S_t \cap S_q|$. Note that $|S_q|$ is a constant for a fixed query topic q.

Definition 3. *A topic t is* analogous *to topic q iff the distance between q and t is less than a given threshold $k \in \mathbb{N}$; i.e., $D(q,t) < k$. That is, t is* analogous *to topic q iff the true impression ($S_t \cap S_q$) is high while the false impression ($S_t \backslash S_q$) is low.*

The goal of this section is to identify a list of topics that are *analogous* to a query topic q. These topics are ranked subsequently based on a *weight function* (Equation 10) that involves both true and false impression values. If any of the *analogous* topics has a bidding cost lower than q, it is a potential alternative for bidding purposes.

Problem 1. *Let q be a given query topic. Identify all topics t that are* analogous *(Definition 3) to q.*

The solution to Problem 1 can be utilized by advertisers to instigate advertising campaigns by choosing the *analogous* topics instead of query topic q, target (approximately) the same set of audiences, and pay less.

Problem 1 could be solved if the target sets for all topics were known. Unfortunately, as explained in Section 2, finding the targets sets is not straightforward (since the I_t sets are unknown). To address this problem, in the rest of this section, we present an approach to identify analogous topics without calculating the exact target sets.

3.1 Properties of analogous topics

The target set of a topic t can be partitioned into two sets: the set of experts E_t and the set of interested users I_t. According to Section 2, the set E_t can be readily identified utilizing the lists. However, I_t is unknown. We aim to identify topics t such that I_t and I_q are "close" (for a suitable definition of close).

We reason about approaches to identify these desired topics. Through this reasoning, we gain some intuition about the properties of analogous topics. Based on the discussion, we conclude this section by introducing two properties of analogous topics that enables us to identify them without calculating the I_t sets.

Approach I: A well-known measure of similarity between two arbitrary sets X and Y is the correlation coefficient, denoted by $\rho(X,Y)$. The correlation between two sets can be calculated utilizing the Pearson product-moment correlation coefficient [15]: $\rho(X,Y) = \frac{cov(X,Y)}{\sigma_X \sigma_Y}$ that is equal to $\frac{n(\sum_{i=1}^n x_i y_i) - \sum_{i=1}^n x_i \sum_{i=1}^n y_i}{\sqrt{(n\sum_{i=1}^n x_i^2 - (\sum_{i=1}^n x_i)^2)(n \sum_{i=1}^n y_i^2 - (\sum_{i=1}^n y_i)^2)}}$ on a sample, where n is the number of elements, and x_i (y_i) is 1 if the i^{th} element belongs to X (Y) and 0 otherwise.

In Theorem 1, we show that there exists a direct translation between the correlation coefficient and the distance of two sets.

Theorem 1. *For any two arbitrary sets X and Y, if the correlation between them is greater than a threshold $\delta \in [-1,1]$, there exist a threshold $k \in \mathbb{N}$, negatively associated with δ ($k \sim -\delta$), such that the distance between X and Y is less than k:*

$$\forall X, Y, \forall \delta \in [-1,1], \\ \exists k \in \mathbb{N}, k \sim -\delta, \rho(X,Y) > \delta \Leftrightarrow D(X,Y) < k \quad (1)$$

Proof Proof Sketch. An increase in $\rho(X,Y)$ is equivalent to an increase in $\sum x_i y_i$ (number of similar items in both sets) that is equivalent to a decrease in $-\sum x_i y_i$ hence a decrease in $D(X,Y)$. Moreover, any increase in δ and subsequently $\rho(X,Y)$ translates to a decrease in $D(X,Y)$ and subsequently k. $\square$

Definition 4. *We define the* correlation *between two arbitrary topics t and t', denoted by $\rho(t,t')$, as the correlation between their target sets; i.e., $\rho(t,t') = \rho(S_t, S_{t'})$. Furthermore, we define the* expertise correlation *between two topics t and t', denoted by $\rho_E(t,t')$, as the correlation between their sets of experts; i.e., $\rho_E(t,t') = \rho(E_t, E_{t'})$.*

According to Theorem 1, for a given query topic q, all topics with a high correlation value with q can be reported as the analogous topics. Since the target sets are unknown, the correlation between two topics cannot be computed. However, we can compute the expertise correlation as follows (the Pearson product-moment correlation coefficient):

$$\rho_E(t,t') = \rho(E_t, E_q) = \frac{cov(E_t, E_q)}{\sigma_{E_t}\sigma_{E_q}}$$
$$= \frac{n(\sum_{i=1}^{n} t_i q_i) - r.s}{\sqrt{(n\sum_{i=1}^{n} t_i^2 - r^2)(n\sum_{i=1}^{n} q_i^2 - s^2)}} \qquad (2)$$

where n is the number of users, and r (s) is the number of expert users on topic t (q). Moreover, t_i (q_i) is 1 if the i^{th} user is expert on topic t (q) and 0 otherwise. The denominator is equal to $\sqrt{(nr - r^2)(ns - s^2)}$. The correlation coefficient can vary from -1 (negatively correlated) to +1 (positively correlated).

A basic approach to approximate the correlation between two topics might be to calculate the expertise correlation between them and to utilize it as a metric to assess the correlation between those topics; i.e., one may claim that $\rho(t,q) \sim \rho_E(t,q)$.

Note that a high expertise correlation between t and q suggests that the distance between E_t and E_q is small (Theorem 1). Thus, among experts, the true impression is large and the false impression is small. The primary idea of Approach I is that if the expertise correlation between t and q is high, one may conclude that the correlation between the whole target sets S_t and S_q is high; hence, according to Theorem 1, the distance between S_t and S_q would be small and t would be analogous to q (Definition 3). Unfortunately, this is not correct as clarified by the following example.

Example 1. *Consider two topics "oil" and "Persian classic dance". Note that as Persians actively argue about both topics (suppose independently in separate posts), many users may place them in lists corresponding to each topic. Therefore, many Persians belong to the expertise sets of both topics, creating a high expertise correlation between these topics. In this sense, we may end up concluding that the topic "oil" is analogous to the topic "Persian classic dance". On the other hand, however, the target sets of these two topics can be very different. A person who is interested in "oil" is not necessarily interested in "Persian classic dance". In other words, by targeting the interested users in one of these topics, we do not target the users interested in the other topic. Thus, a high correlation between sets of experts does not imply the same for the corresponding sets of interested users; therefore the target sets for these topics are not necessarily related and* $\rho(t,q) \not\sim \rho_E(t,q)$.

This problem may be resolved by not looking at topics "in isolation" but in conjunction with other topics. The two topics "oil" and "Persian classic dance" have high expertise correlation. However, let us consider other topics with high expertise correlation to "oil" or "Persian classic dance". The topic "oil" has high expertise correlation to topics in $\mathcal{S}_{oil} = \{energy, power, war, \cdots\}$ for example, whereas "Persian classic dance" has high expertise correlation to topics in $\mathcal{S}_{dance} = \{art, music, culture, \cdots\}$. The expertise correlation between topics in $\mathcal{S}_{oil}$ and $\mathcal{S}_{dance}$ is extremely low.

Example 1 suggests that a holistic view, that considers the expertise correlation of all topics in conjunction rather than individual topics in isolation, might help to determine

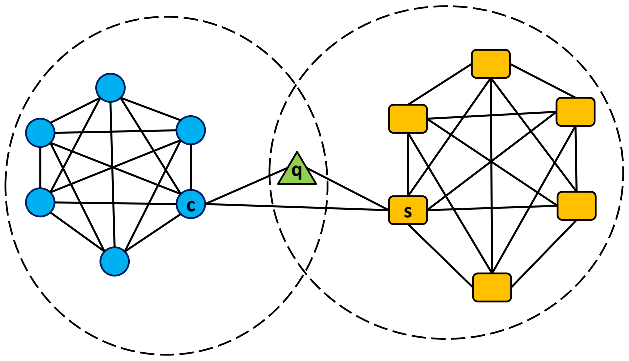


Figure 1: Partitioning a graph of topics. Any optimal clustering algorithm would generate two clusters as shown: node q may be assigned to each of the two clusters. Note that an optimal clustering will not generate a cluster $\{q, c, s\}$.

topics that are analogous to q. It is a natural tendency of users to be interested in high-level topic categories as well as topics under these categories. For example, if user u is interested in Wimbledon (the tennis tournament), it is natural to assume, that with a high probability, u is interested in the bigger category tennis as well as other tennis events such as French Open, US Open, Australian Open, etc. If u is interested in Oscars, it is safe to assume, with a high probability, u is also interested in other film events such as Golden Globe, BAFTA, Cannes film festival, Berlin film festival, etc. Based on this, we can conclude that for topics in the same category (e.g., Wimbledon and US Open which are both tennis events), the sets of interested users are close (i.e., if t_1 and t_2 are members of the same category of topics, the distance $D(I_{t_1}, I_{t_2})$ is small). This suggests that identifying the topics in the category that topic q belongs would aid in locating the analogous topics.

Approach II: One approach to incorporate this holistic view might be to calculate the expertise correlation between all topics and create a correlation graph where nodes represent topics and the weight of an edge between two arbitrary nodes t and t' is $\rho_E(t,t')$. Then partition (or classify) this graph and report all topics in the partition containing topic q, as the topics analogous to q. Unfortunately, this approach has its own shortcomings as shown in the following example.

Example 2. *Consider the graph shown in Figure 1. Each node in this graph represents a topic with node q being the query topic. All the nodes represented by a circle have a high expertise correlation with each other, and all nodes represented by a square have a high expertise correlation with each other. The expertise correlation is small between a circle and a square node. Node q has a high expertise correlation with nodes c and s, and there is a high expertise correlation between nodes c and s.*

If we are looking for topics analogous to q, ideally one should identify c and s. However, any clustering scheme that relies on a global objective function based on the expertise correlations will partition this graph into two clusters as shown in Figure 1 without returning $\{q, c, s\}$ *as a separate cluster. We note that for any algorithm that generates a given number of partitions k, one can generalize this example by creating sets of k different shapes, without changing the behavior observed.*

The problem encountered in Example 2 is a result of the fact that clustering algorithms rely on the optimization of a global objective function that does not take into account the original topic of interest q. Assigning q to a cluster takes place based on the optimality of a global function, and that can lead to poor performance in situations where the focus is solely on q.

Conclusion: These two examples suggest that one needs a hybrid approach considering both the direct expertise correlation between each topic and q, as well as the expertise correlation amongst the neighbors.

A topic t is analogous to topic q if and only if:

PROPERTY 1: The expertise correlation between q and t is greater than a threshold δ; i.e., $\rho_E(q,t) > \delta$.

PROPERTY 2: Topic t is in the same category of topics as topic q. In other words, the topics having high expertise correlation with topic t should have high expertise correlation with topic q and vice versa:

$$\forall t'; \rho_E(t,t') > \delta \Leftrightarrow \rho_E(q,t') > \delta$$

When property (1) is satisfied, the distance between E_t and E_q is small (Theorem 1). Moreover, property (2) suggests that the distance between I_t and I_q is small. Therefore, when both properties are satisfied, the distance between the target set of topic t ($S_t = E_t \cup I_t$) and the target set of topic q ($S_q = E_q \cup I_q$) is small; thus t is analogous to q.

In most real world scenarios, it is impossible to identify topics t that strictly satisfy both properties. Therefore, we introduce a technique in Section 3.2 that considers both properties, by defining a "trade-off" between them. In other words, our approach assigns weights to any topic t based on the direct expertise correlation between t and q (Property 1), and at the same time penalizes that weight (by associating a cost) if the topics having high expertise correlation with t have low expertise correlation with q, or the topics with low expertise correlation with t have high expertise correlation with q (Property 2).

3.2 Computing analogous topics

Recall that $U = \{u_1, u_2, \cdots, u_n\}$ denotes the set of users and $T = \{t_1, t_2, \cdots, t_m\}$ denotes the set of topics.

DEFINITION 5. *The* expert coverage probability *of topic $t \in T$, denoted by $\mathcal{P}(t)$, is the fraction of users in U that are expert on t. In particular, $\mathcal{P}(t) = \frac{|E_t|}{|U|}$. Moreover, the* expert coverage probability *of topic $t \in T$* given *topic q (the* conditional expert coverage probability, *denoted by $\mathcal{P}(t|q)$) is the fraction of users in $U' = E_q$ that are expert on t. In particular, $\mathcal{P}(t|q) = \frac{|E_t \cap E_q|}{|E_q|}$.*

As an example, suppose U consists of 1000 users, among them 10 users are expert on "drawing" and 50 users are expert on "music". This leads to $\mathcal{P}(drawing) = 10/1000 = 0.01$ and $\mathcal{P}(music) = 0.05$.

For any two topics t and q, the probabilities $\mathcal{P}(t|q)$ and $\mathcal{P}(t)$ may be significantly different. As an example, assume we observe that among experts on topic "Picasso", 60% are expert on "drawing" and 5% are expert on "music". Thus, $\mathcal{P}(drawing|Picasso) = 0.6 >> P(drawing)$ while $\mathcal{P}(music|Picasso) = \mathcal{P}(music) = 0.05$ (showing that music and Picasso are independent topics). We argue that these changes in the expert coverage probability of different topics given a fixed topic can be utilized as an equivalent measure to Property 1.

We utilize a two-state automaton to study whether any topic t is analogous to a given query topic q or not. This automaton has two states $\mathcal{N}$ and $\mathcal{A}$ corresponding, respectively, to the concepts of "Not-analogous" (t and q are not analogous) and "Analogous" (t and q are analogous). Given topic q, while considering topic t, the automaton can be in one of the states $\mathcal{N}$ or $\mathcal{A}$. The $\mathcal{N}$ state corresponds to low conditional expert coverage probability and the $\mathcal{A}$ state corresponds to high conditional expert coverage probability. These states determine how far $\mathcal{P}(t|q)$ is from the original $\mathcal{P}(t)$ assessing whether topic t satisfies Property 1 (Theorem 2). For any topic t, we aim to identify the state of the automaton with the maximum likelihood.

We deploy a binomial distribution as the basis to realize such measurement. The binomial distribution is a density function that determines the probability that r successes are achieved in a sequence of d independent experiments, when a success is yield with a fixed probability p. In the case of topics and experts, this expresses the probability that among d experts on q, r users are expert on a topic t where $p = \mathcal{P}(t)$. Adhering to the binomial distribution, the probability that the automaton is in state $\mathcal{N}$ for a topic $t \in T$ is:

$$\mathcal{P}(\mathcal{N}_t|q) = \frac{\binom{d}{r_t}\mathcal{P}(t)^{r_t}(1-\mathcal{P}(t))^{d-r_t}}{Z} \tag{3}$$

where $r_t = |E_t \cap E_q|$, $d = |E_q|$. Similarly the probability that the automaton is in state $\mathcal{A}$ is:

$$\mathcal{P}(\mathcal{A}_t|q) = \frac{\binom{d}{r_t}(\alpha \times \mathcal{P}(t))^{r_t}(1-\alpha \times \mathcal{P}(t))^{d-r_t}}{Z} \tag{4}$$

where $\alpha > 1$ (a constant), and $\alpha \times \mathcal{P}(t)$ is the expected expert coverage probability of t given q in case t is analogous to q. Here, $Z = \binom{d}{r_t}\mathcal{P}(t)^{r_t}(1-\mathcal{P}(t))^{d-r_t} + \binom{d}{r_t}(\alpha \times \mathcal{P}(t))^{r_t}(1-\alpha \times \mathcal{P}(t))^{d-r_t}$ is a normalizing constant. Since the denominator Z is similar in both equations and does not impact the calculations, hereafter, we ignore it and just consider the numerators in calculating and comparing $\mathcal{P}(\mathcal{A}_t|q)$ and $\mathcal{P}(\mathcal{N}_t|q)$.

THEOREM 2. *The value of $\frac{\mathcal{P}(\mathcal{A}_t|q)}{\mathcal{P}(\mathcal{N}_t|q)}$ increases (decreases) iff the distance $D(E_t, E_q)$ decreases (increases).*

PROOF. Let α be fixed. The value of $\frac{\mathcal{P}(\mathcal{A}_t|q)}{\mathcal{P}(\mathcal{N}_t|q)}$ increases (decreases) when $\alpha^{r_t}(\frac{1-\alpha\mathcal{P}(t)}{1-\mathcal{P}(t)})^{d-r_t}$ increases (decreases). The latter increases (decreases) when r_t increases (decreases) or $\mathcal{P}(t)$ decreases (increases). This is because $\alpha > 1$ and $0 < \frac{1-\alpha\mathcal{P}(t)}{1-\mathcal{P}(t)} < 1$. Moreover $\mathcal{P}(t)$ decreases (increases) when $|E_t|$ decreases (increases). In both cases $D(E_t, E_q)$ decreases (increases). $\square$

To incorporate Property 2, we create a *correlation graph*: a graph $G = (M, E)$ where any topic $t \in T - \{q\}$ corresponds to a node in G. Moreover, for any two nodes $m_i, m_j \in M$ representing topics t_i and t_j, the weight of the edge e connecting m_i and m_j is $w_e = w(m_i, m_j) = \rho_E(t_i, t_j)$.

DEFINITION 6. *Suppose the state of the automaton for any topic is determined. In particular, the automaton is in state s_i ($\mathcal{N}$ or $\mathcal{A}$) when considering topic t_i (t_i corresponds to node m_i in G). The edge $e = (m_i, m_j)$ is called* inconsistent *if ($w_e > 0$ and $s_i \neq s_j$) or ($w_e < 0$ and $s_i = s_j$).*

Definition 6 suggests that an edge $e = (m_i, m_j)$ is *inconsistent* if the automaton is in different states when the expertise correlation between topics t_i and t_j (corresponding to nodes m_i and m_j) is positive ($\rho_E(t_i, t_j) > 0$) or when the automaton is in the same state if $\rho_E(t_i, t_j) < 0$.

Problem 2 utilizes the automaton and the correlation graph to identify the most likely states for all topics maximizing $\prod_{t_i \in T-\{q\}} \mathcal{P}(s_i|q)$ (or equivalently $\sum_{t_i \in T-\{q\}} \log \mathcal{P}(s_i|q)$) where $s_i \in \{\mathcal{N}, \mathcal{A}\}$ is the state assigned to topic t_i. To satisfy Property 2, Problem 2 associates a cost with any inconsistent edge.

Problem 2. *Let $G = (M, E)$ be the correlation graph for topics in $T - \{q\}$. Identify the state of the automaton for each node $m_i \in M$ ($s_i \in \{\mathcal{N}, \mathcal{A}\}$) to*

$$maximize \sum_{m_i \in M} \log \mathcal{P}(s_i|q) - \sum_{e \in E} c_e \qquad (5)$$

Note that c_e is the cost of edge $e = (m_i, m_j)$ that is equal to $|w_e|$ if e is inconsistent or zero otherwise. By adding a constant factor $\sum_{\substack{e \in E \\ w_e < 0}} w_e$ to Equation 5, we get

$$(5) \equiv maximize \sum_{m_i \in M} \log \mathcal{P}(s_i|q) \; - \sum_{\substack{e=(m_i,m_j) \in E \\ s_i \neq s_j}} w_e \qquad (6)$$

Maximizing Equations 5 and 6 is equivalent to maximizing the probability that the correlation graph G is created by a two-state automaton where the probability that the automaton is in state $\mathcal{N}$ or $\mathcal{A}$ for each node in G is derived by Equations 3 and 4 (corresponding to Property 1) and for each edge e in G, the probability that the automaton maintains the same state over the two end-points of that edge depends on w_e (corresponding to Property 2).

Theorem 3. *Problem 2 is NP-hard.*

Proof. We reduce the max-cut problem to Problem 2. In max-cut problem, given a weighted graph G, the goal is to partition vertices of G into two subsets S_1 and S_2 such that the weight of edges between S_1 and S_2 is maximized. The max-cut problem is widely known to be NP-hard [12].

The reduction is as follows. Let us assume we want to identify the maximum cut for the graph G. We create a graph G' where there is a node u' in G' for any node u in G. For any edge $e = (u_i, u_j) \in G$, we add an edge $e' = (u'_i, u'_j)$ in G' (u'_i and u'_j are the nodes in G' corresponding to u_i and u_j in G) with a weight of $w_{e'} = -w_e$ where w_e is the weight of edge e in graph G. Moreover, for any two nodes v_i and v_j in G not connected to each other, we add an edge between their corresponding nodes v'_i and v'_j in G' with a weight of $w_{e'} = 0$. Finally, we set the probability that the automaton is in the $\mathcal{N}$ or $\mathcal{A}$ state for any node in G' to be equal. Identifying the maximum cut for graph G reduces to solving Problem 2 for graph G' by identifying the state of automaton for any node in G' that maximizes Equation 6 that is equivalent to maximizing $W = \sum_{e'=(u'_i,u'_j) \in E'; s_{u'_i} \neq s_{u'_j}} (-w_{e'})$.

After identifying the optimal states of the automaton for each topic, we define the set S_1 containing all nodes in G corresponding to nodes in G' that are assigned with a $\mathcal{N}$ state and S_2 containing all nodes in G corresponding to nodes in G' that are assigned with a $\mathcal{A}$ state. Thus, $W = \sum_{e \in E'} w_e$ where E' contains all edges in E with an endpoint in S_1 and the other endpoint in S_2. In this sense, maximizing W is equivalent to identifying the maximum cut. □

To identify analogous topics, a more general approach would be to model this process by a 3-state automaton. The automaton, for any topic, can be in any of the 3 states "Dissimilar", "Independent", or "Analogous". The conditional expert coverage probability of topic t on these states is, respectively, a_1, a_2 and a_3 where $a_1 < a_2 < a_3$. In particular, $a_2 = \mathcal{P}(t)$ is the expert coverage probability for topic t over U, a_1 is a lower conditional expert coverage probability showing that the topics t and q are dissimilar (perhaps negatively analogous), and a_3 is a higher conditional expert coverage probability showing that the topics t and q are analogous. In the present work, we simplify the model and merge the "Dissimilar" and "Independent" states to form a "Not-analogous" state $\mathcal{N}$. This generalization can be conducted easily following the developments in the section.

3.3 IAT: an algorithm to Identify Analogous Topics

According to Theorem 3, Problem 2 is NP-hard. It involves two parts: (1) maximizing the log-likelihood of expert coverage probabilities over all nodes; i.e., $\sum_{m_i \in M} \log \mathcal{P}(s_i|q)$, and (2) minimizing the cost of inconsistent edges; i.e., $\sum_{e \in E} c_e$. The value for the first part can be calculated for each node independently. Computing the second part, however, needs to be aware of the states of the neighboring nodes.

We propose a technique (called IAT) that adopts a heuristic approach to reduce the complexity of Problem 2. The main root of the complexity in Problem 2 is the existence of cycles in the graph. In an acyclic graph, we can order nodes of the graph and identify the best state assignment by optimizing both parts of Equation 5 node-to-node based on this ordering. However, when the graph contains a cycle, no ordering can be assumed between nodes in the cycle; the states of all these nodes depend on each other (due to the second part) and should be determined simultaneously. This leads to a complex structure to deal with. The basic idea of IAT is to obtain an acyclic subgraph (a spanning tree) of the original graph. We, then, identify the optimal states based on this tree. Our experiments show that by utilizing this technique, we can effectively locate the analogous topics.

This approach raises the question on how to choose the spanning tree. In Problem 2, before determining the state of a node, we consider the states of the neighboring nodes in order to reduce the cost of inconsistent edges. Among all edges connected to an arbitrary node u, some have the highest probability to be inconsistent. We refer to these as *inconsistency-prone* edges. The goal is to assign the states such that (1) the log-likelihood of expert coverage probabilities over all nodes is maximized and (2) the cost of inconsistency-prone edges is minimized. The idea is that since the inconsistency-prone edges are the most likely edges to induce costs, a state assignment that reduces the cost over these edges, reduces the cost over all edges.

To locate the inconsistency-prone edges, we define an *expected cost* value for each edge. The edges with high expected cost values are considered inconsistency-prone. Let $\hat{A}_u = \frac{\log \mathcal{P}(\mathcal{A}_u|t)}{\log \mathcal{P}(\mathcal{A}_u|t) + \log \mathcal{P}(\mathcal{N}_u|t)}$ determine the expected probability that u is associated with state $\mathcal{A}$ and $\hat{N}_u = 1 - \hat{A}_u$ be the expected probability that u is associated with state $\mathcal{N}$.

The *expected cost* of an edge $e = (u, v)$ denoted by $\hat{c}_e$ is:

$$\hat{c}_e = \begin{cases} |\hat{A}_u - \hat{A}_v| \times w_e & \text{if } w_e \geq 0, \\ (1 - |\hat{A}_u - \hat{A}_v|) \times |w_e| & \text{if } w_e < 0. \end{cases} \quad (7)$$

where the value $|\hat{A}_u - \hat{A}_v|$ (a value between 0 and 1) determines the difference in probability of being associated with state $\mathcal{A}$ for adjacent nodes u and v. High values of $|\hat{A}_u - \hat{A}_v|$ suggest that u and v are likely to be assigned with different states. Having the expected cost values, Problem 3 identifies the optimal acyclic subgraph.

PROBLEM 3. *Considering the expected cost of each edge in the graph $G = (M, E)$, identify an acyclic subgraph $T = (M, E^*)$ with maximum sum of the expected costs over all edges in E^*.*

Problem 3 is equivalent to the minimum spanning tree problem. We can create a new graph G' by negating the weights of all edges in G and identifying the minimum spanning tree in G'. This tree would be the optimal solution for Problem 3 that can be found utilizing any MST algorithm such as Kruskal [11] or Prim [4]. The run time complexity for these algorithms on a dense graph is $O(m^2)$ where m is the cardinality of M.

Assume tree T is the optimal solution for Problem 3. The IAT algorithm is a dynamic programming approach that calculates two values $LP(\mathcal{N}_u)$ and $LP(\mathcal{A}_u)$ for any node $u \in M$ starting from leaves, going upwards to the root. For each leaf u, $LP(\mathcal{N}_u) = \log \mathcal{P}(\mathcal{N}_u|q)$ and $LP(\mathcal{A}_u) = \log \mathcal{P}(\mathcal{A}_u|q)$ that are calculated based on Equations 3-4. For any inner node u, the values are calculated as follows:

$$LP(\mathcal{A}_u) = \log \mathcal{P}(\mathcal{A}_u|q) + \sum_{v \in C(u)} \max(LP(\mathcal{A}_v), LP(\mathcal{N}_v) - w(u,v)), \quad (8)$$

$$LP(\mathcal{N}_u) = \log \mathcal{P}(\mathcal{N}_u|q) + \sum_{v \in C(u)} \max(LP(\mathcal{A}_v) - w(u,v), LP(\mathcal{N}_v)), \quad (9)$$

where $C(u)$ is the set of u's children and $w(u, v)$ is the weight of edge (u, v).

When all values are calculated, IAT identifies the best state assignment to all nodes by locating the chain of states maximizing the value of $\max(LP(\mathcal{A}_r), LP(\mathcal{N}_r))$ where r is the root of the T.

The pseudo-code for IAT is presented as Algorithm 1. Note that p_u is the parent of u and $C(u)$ is the set of u's children in Tree T. The variable $APointer_u$ ($Npointer_u$) saves the optimal state assigned to u when its parent p_u is assigned with a state $\mathcal{A}$ ($\mathcal{N}$). The function "arg max(a, b)" returns $\mathcal{A}$ if $a > b$ and returns $\mathcal{N}$ otherwise. Finally, s_u holds the assigned state of node u. IAT reports all topics that are assigned with state $\mathcal{A}$ as the analogous topics. For each analogous topic t, we define a weight as

$$weight(t) = \log \mathcal{P}(\mathcal{A}_t|q) - \log \mathcal{P}(\mathcal{N}_t|q) \quad (10)$$

This weight determines the improvement we achieve when topic q is assigned with state $\mathcal{A}$ instead of $\mathcal{N}$. Thus, topics with higher weights correspond to more prominent relationships with topic q. Algorithm IAT ranks the analogous topics based on these weight values and returns Q, a ranked list of all nodes assigned with a $\mathcal{A}$ state.

Algorithm 1: The IAT algorithm

```
   input  : The correlation graph $G = (M, E)$, Topic $q$
   output: A ranked list of analogous topics $Q$
   // Identify the optimal spanning tree
1  Calculate the expected cost of edges according to Eq. 7
2  Identify the optimal spanning tree $T$ (e.g., by Prim)
   // Probability calculations
3  Traverse $T$ bottom-up (from leaves to the root):
4  foreach $u \in M$ do
5      $LP(\mathcal{A}_u) = \log \mathcal{P}(\mathcal{A}_u|q) + \sum_{v \in C(u)} \max(LP(\mathcal{A}_v), LP(\mathcal{N}_v) - w(u,v))$
6      $LP(\mathcal{N}_u) = \log \mathcal{P}(\mathcal{N}_u|q) + \sum_{v \in C(u)} \max(LP(\mathcal{A}_v) - w(u,v), LP(\mathcal{N}_v))$
7      $APointer_u = \arg\max(LP(\mathcal{A}_u), LP(\mathcal{N}_u) - w(u, p_u))$
8      $NPointer_u = \arg\max(LP(\mathcal{A}_u) - w(u, p_u), LP(\mathcal{N}_v))$
   // Identifying the state of the root
9  $s_r = \arg\max(LP(\mathcal{A}_r), LP(\mathcal{N}_r))$
   // Identifying the state for all nodes
10 Traverse $\mathcal{F}$ top-down (from roots to leaves):
11 foreach $u \in M$ do
12     $s_u = NPointer_u$;
13     if $s_{p_u}$ = "$\mathcal{A}$" then
14         $s_u = APointer_u$
   // Sort the analogous topics
15 $Q = \emptyset$
16 foreach $u \in M$ do
17     if $s_u$ = "$\mathcal{A}$" then
18         $Q = Q \cup \{u\}$
19 Sort $Q$ based on Eq. 10
```

THEOREM 4. *The IAT algorithm identifies the optimal state assignment on the tree. The run time complexity of IAT is $\theta(m^2)$ where m is the number of topics.*

PROOF PROOF SKETCH. IAT is a standard dynamic programming approach that solves Problem 2 step by step from leaves to the root. The value $LP(\mathcal{A}_u)$ is the optimal solution for Problem 2 on the subtree rooted at u when the state of u is $\mathcal{A}$. Similarly $LP(\mathcal{N}_u)$ is the optimal solution for Problem 2 on the same subtree when the state of u is $\mathcal{N}$. Therefore the value of $\max(LP(\mathcal{A}_r), LP(\mathcal{N}_r))$ is the optimal solution for the whole tree.

We can calculate the expected cost of each edge in constant time. Since there are m^2 edges, line 1 takes $\Theta(m^2)$. Prim's algorithm implemented with Fibonacci heap takes $\Theta(m^2)$ to identify the MST. The probability calculation phase takes $\Theta(m)$ since each edge in the MST can update LP of one node only once. The state identification phase also takes $\Theta(m)$ to calculate the optimal states for all nodes. Finally it takes $\Theta(m \log m)$ to sort the analogous topics. Thus, in total IAT takes $\Theta(m^2)$. □

4. CATEGORIZING THE FOLLOWERS

In Section 1 we explained it is very helpful to categorize all experts on a given topic q based on other topics of their expertise in order to engage them in word of mouth campaigns. For example, among all experts on *social media*, those who are expert on topics such as "consumer behavior", "distribution channel", "market-based pricing", "sales", etc. can be potentially categorized together (in a big category of "Marketing"); and those expert on topics such as "high ranking placement", "website visitors", "Google results", "search engine traffic", "white hat seo", etc. can form a big category of "Search Engine Optimization".

By categorizing the experts, we would be able to understand them in a more refined fashion and to locate the experts that are the "right" advocates to instigate a popularity propagation (based on word of mouth effects) in the network.

4.1 CTE: an algorithm to Categorize Topics and Experts

Let q be a topic, E_q be the set of experts on q, and T_u be the set of all topics user u is an expert on. We propose an algorithm (called CTE) to categorize users $u \in E_q$ based on the topics of their expertise. We introduce four desirable properties that CTE should have:
(1) Soft clustering: Users may be assigned to several categories. This is desirable as users usually have diverse topics of expertise hence they might belong to various categories.
(2) Unknown number of categories k: The optimal number of categories is unknown. The algorithm should identify the best number of categories instead of requiring it as an input.
(3) Coping with high dimensional data: The number of topics is large. On high dimensional datasets, any approach based on distance (e.g., the traditional clustering algorithms) is inaccurate since distances between all pairs converge.
(4) Considering the correlation between topics: Topics are correlated; any approach that is based on an assumption that dimensions (topics in this case) are independent is not applicable.

In Sections 5 and 6, we argue that traditional clustering algorithms fail to provide useful categorizations. Here, we present an approach (satisfying the properties above) that considers topics and users in two steps: first it categorizes the topics without taking into account the users (topic categorization phase); and then it assigns each user $u \in E_q$ based on T_u, to the topic categories (user assignment phase).

This separation of topics and users in categorization helps to segment topics into partitions that are representing high-level topic categories. When we utilize an approach that simultaneously categorizes users in E_q and topics in $\bigcup_{u \in E_q} T_u$ (e.g., the bi-clustering techniques), topics are categorized according to the correlations calculated utilizing the sets T_u of users u in E_q, instead of utilizing the sets T_u of all users in U. Incorporating the users in E_q (instead of *all* users) to capture correlations introduces *coverage bias*.

Coverage bias loosely means that users in E_q are not representative of the population. There are cases where the correlation between two topics t_1 and t_2 is low but the topics are highly correlated in the context of a query topic q (i.e., based on users in E_q). For example, consider topics "Queen's park" and "Government" in the context of topic "Ontario". These topics are not highly correlated in general. However, when the set we consider consists of experts on "Ontario", the two topics would be highly correlated, since Queen's park is the home for the Legislative Assembly of Ontario and is usually utilized as a metonym for the Government of Ontario. On the other hand, there are cases where two topics t_1 and t_2 are highly correlated but when considered in the context of experts on a query topic q, this correlation is small. Consider two topics "football" and "rugby" given the query topic "fifa" as an example. In general rugby and football are correlated due to the relation between rugby and the American football. However, given the topic "fifa", the term "football" would usually refer to the international "football" that has low correlation with "rugby".

4.1.1 *Topic categorization*

The CTE algorithm runs in two phases: (1) topic categorization, and (2) user assignment. Topic categorization starts by creating the correlation graph among topics as discussed in Section 3.2 (incorporating all users in U in weight calculations). Subsequently, we aim to segment topics (graph nodes) into categories such that topics with positive expertise correlation values are located in the same category and topics with negative expertise correlation values are located in different categories.

Problem 4. *Let* $G = (V, E)$ *be a* correlation graph *where topic* $t \in \bigcup_{u \in E_q} T_u$ *corresponds to a node in* V*. Also, the weight of the edge connecting any pair of nodes* $u_i, u_j \in V$ *(representing topics* t_i *and* t_j*) is* $w_{u_i u_j} = \rho_E(t_i, t_j)$*. Segment* G *into categories such that the sum of the weights of edges with positive weights that are cut and edges with negative weights that are uncut is minimized.*

Bansal et. al. have shown that Problem 4 is NP-hard even for a simple case where the weight of all edges are either -1 or $+1$ [1]. Demaine et. al. have shown that Problem 4 and the weighted multicut problem are equivalent; Problem 4 is APX-hard; and obtaining any approximation bound better than $\theta(\log n)$ is difficult ($n = |V|$). Utilizing the linear programming rounding and "region growing" techniques, they have proposed an algorithm to approximate Problem 4 with a tight bound of $\theta(\log n)$ [5].

This approach models the problem as a linear program. A zero-one variable x_{uv} is defined for any pair of vertices u and v. The equation $x_{uv} = 0$ suggests that u and v are in the same category; $x_{uv} = 1$ declares the opposite. Problem 4 translates to

$$\textit{minimize} \sum_{(u,v):\ w_{uv}<0} |w_{uv}|(1 - x_{uv}) + \sum_{(u,v):\ w_{uv}>0} |w_{uv}| x_{uv}$$

subject to the following constraints:
(1) $x_{uv} \in [0, 1]$, (2) $x_{uv} = x_{vu}$, and (3) $x_{uv} + x_{vw} \geq x_{uw}$.
A "region growing" technique is adopted, afterwards, to transform the fractional values of x_{uv} to integral values 0 or 1. The basic idea is to grow balls around graph nodes (with a fixed maximum radius). Each ball is reported as a category. Therefore, two nodes u and v with a high value x_{uv} would be assigned to two different balls and finally two different categories (equivalent to setting $x_{uv} = 1$).

The run time complexity of the algorithm proposed by Demaine et. al. is $\mathcal{O}(n^7)$. This approach is not practical for datasets containing a large number of topics. In our implementation of topics based on Twitter lists, we construct millions of topics for Twitter users. Any approach based on an $\mathcal{O}(n^7)$ algorithm is deemed not practical for our setting.

We propose a heuristic approach called MaxMerge to categorize the correlation graph when the graph is large. The CTE algorithm utilizes MaxMerge in the topic categorization phase. To start, MaxMerge constructs a category for each vertex in G. The algorithm proceeds iteratively. In each iteration, it calculates the value Δ_{AB} achieved by merging any pair of existing categories A and B. The value Δ_{AB} is the average of the weights of edges with one end-point in category A and one end-point in category B. According to Problem 4, the objective is to categorize G such that the edges with positive weights are in the same category and the edges with the negative weights are amongst different categories. The Δ_{AB} value expresses our progress towards the objective when the two categories are merged. At each iteration, categories A and B having the maximum positive value of Δ_{AB} will be merged; MaxMerge continues as long as this maximum positive value is greater than the average weight of all node pairs in the whole graph (stating that merging the two categories at hand should result in a value that is higher than the average weight of one big category that includes all topics). Pseudo code of MaxMerge is provided as Alg 2. The input is a graph $G = (V, E)$.

Algorithm 2: The MAXMERGE algorithm

```
1 Let $avr$ be the average of the weight of all edges in $E$
2 Consider each node as a category
3 foreach pair of categories $A$ and $B$ do
4     $SUM = \sum$ weight of all edges between A and B
      $\Delta_{AB} = SUM/(|A| * |B|)$
5 $Max = \max_{A,B} \Delta_{AB}$;   $A^*, B^* = \arg\max_{A,B} \Delta_{AB}$
6 if $Max > avr$ then
7     Merge $A^*$ and $B^*$ to one category and Goto step 3
8 return all categories
```

Theorem 5. *The run time complexity of Algorithm 2 is $\mathcal{O}(m^2 \log m)$ where $m = |V|$.*

Proof. Line 1 takes $\mathcal{O}(m^2)$ since there are m^2 edges between the topics. Line 2 takes $\mathcal{O}(m)$. At the beginning there are $\mathcal{O}(m^2)$ pairs of partitions and it takes $\mathcal{O}(1)$ to calculate Δ values for each pair. Thus it take $\mathcal{O}(m^2)$ to calculate these values at the first iteration. We can store these values in a priority queue. Based on the implementation it takes $\mathcal{O}(m^2)$ or $\mathcal{O}(m^2 \log m)$ to create this priority queue.

We do several iterations while $Max > avr$ to merge the partitions. The number of iterations is at most $m - 1$. In each iteration, it takes $\mathcal{O}(\log m)$ to find and delete the max value, $\mathcal{O}(m)$ to merge the two partitions, $\mathcal{O}(m)$ to update the values of Δ for the new merged partition, and $\mathcal{O}(m \log m)$ to update these values in the priority queue. Note that if we merge two partitions A and B into the new partition C, for any partition D: $SUM_{CD} = SUM_{AD} + SUM_{BD}$.

Therefore, overall the run time is $\mathcal{O}(m^2 \log m)$. □

4.1.2 Assigning the experts

Once the topic categories are identified, we assign users in E_q to these categories. CTE assigns a user $u \in E_q$ based on T_u (Algorithm 3): it assigns u to any category containing at least one topic in T_u. Note that adhering to this approach, a user u can be a member of several partitions expressing u's diversified expertise on various high-level topics.

5. EXPERIMENTS

We evaluate the proposed algorithms IAT and CTE on a dataset containing about 4.5 million lists (that is all lists available in Twitter when we collected the data). Each list l_i is associated with a topic t_i (hence 4.5 million topics)[1] For each user u in a list l_i, the corresponding topic t_i is considered as a topic of expertise for u (i.e., $t_i \in T_u$). There are 13.5 million distinct users in these lists.

Algorithm 3: Expert Assignments

```
1 Create an empty category $\tilde{C}$ for each topical category $C$
2 foreach user $u$ in $E_q$ do
3     foreach topic category $C$ do
4         if there exist topic $t \in C$ such that $t \in T_u$ then
5             $\tilde{C} = \tilde{C} \cup \{u\}$
6 Output all categories $\tilde{C}$
```

Table 1: The Impact of pruning on the number of topics.

Query topic	number of experts	number of topics	**size**: number of topics after the 1% pruning
social+media	375809	1360060	551
canada+politics	460	19080	1490
wine+Toronto	1337	27061	938
cloud+computing	56	2769	2769
fashion+trends	1112	36263	886

We execute the algorithms on a machine with a 16 core AMD *Opteron*^{TM} 850 Processor. This machine runs CentOS 5.5 (kernel version 2.6.18-194.11.1.e15) and contains 100GB of memory. All algorithms are single-threaded and are implemented in Java.

We observe similar trends when evaluating our algorithms with different query topics. Here, we report results for the following 5 queries: (1) canada+politics, (2) cloud+computing, (3) social+media, (4) toronto+wine, and (5) fashion+trends. For each query q (e.g., social+media), we retrieve all users whose topics of expertise match each input query (e.g., all users who are expert on social and also on media). These users form the set of experts E_q for the given query q.

5.1 Identifying the analogous topics

Identifying the analogous topics for the aforementioned queries involves two steps: (1) creating the correlation graph, and (2) assigning $\mathcal{A}$ or $\mathcal{N}$ states to topics (Algorithm IAT).

Figure 2(a) shows the distribution of topics and experts for query social+media. We count how many users in E_q (for a given query q) are expert in each topic in $\bigcup_{u \in E_q} T_u$. We see a similar trend for all other queries. This figure suggests that the curve displaying the number of experts on each topic has a heavy tail. Thus, pruning the topics with very small frequency can significantly help in improving performance. The run time of identifying analogous topics for the query social+media is measured utilizing different pruning percentages and reported in Figure 2(b). Here, pruning with a percentage of α means that the topics that appear in the expertise sets of less than α% of the experts are removed; i.e., a topic t is pruned iff $\frac{|\{u \in E_q | t \in T_u\}|}{|E_q|} < \frac{\alpha}{100}$.

We see that a pruning of only $1 - 2\%$ can significantly decrease the run time. On the other hand, pruning does not have a major impact on the accuracy of the results. The

[1]The precise process we follow to make this association can be found in [2].

Table 2: Analogous topics (topics are presented stemmed)

	social media	canada politics	toronto wine	cloud computing	fashion trends
1	busi	polit	food	tech	fashion
2	entrepreneur	news	food wine	cloud	trendsett
3	pr	canada	foodi	cloud comput	blogger
4	polit	politicsdemocraci economi	food drink	technolog	blog
5	journalist	canadian polit	restaur	a68	fashion blogger
6	seo	media	canada	cloudcomput	fashionista
7	entertain	news polit	wine	cloudyp	fashion beauti
8	info	cdnpoli	toronto food	tech news	media
9	internet market	politico	canadian	cloud 0	design
10	communic	peopl	eat	cloud virtual	fashion blog
11	industri	canadian	toronto restaur	virtual	shop
12	advertis	local	chef restaur	news	news
13	fav	progress	media	busi	creativ
14	brand	toronto	resto	vendor	lifestyl
15	engag	journalist	food toronto	techi	fashion style
16	communiti	blogger	toronto foodi	work	beauti
17	inform	interest	we like eat drink	softwar	beauti fashion
18	onlin market	cdn polit	all	cloud saa	busi
19	digit market	liber	ontario	cloudcomputingenthusiast	inspir
20	cultur	govern	culinari	clouderati	entertain

topics that are reported as analogous are very similar for all these pruning percentages (e.g., no difference exists in the top 10 topics when we prune the topics with various percentages 0.1%-10%). We observe similar behavior for all other queries. For the rest of this section, we use a pruning of 1% of the topics to improve performance. Table 1 shows the number of topics that are not pruned in this step for the given five queries. According to Table 1, the pruning step with even a very small value of 1% significantly reduces the dimensionality of the problem. Hence, the problem can be solved more efficiently. The only exception here is the query for cloud+computing. We note that this query has 56 experts. A pruning of 1% removes any topic appearing in the expertise set of less than 0.56 users; thus, no topic is removed in this case (all topics appear in the expertise set of at least 1 user).

Table 2 reports the top-20 topics for each query as identified by IAT. The analogous topics are sorted based on Equation 10. In Table 2, we observe, for example, that topics such as "busi(ness)" (topics are presented stemmed), "entrepreneur", "journalist", "seo", "internet market(ing)", and "communic(ation)" are analogous to the query social+media.

The utility of this information is evident: instead of focusing on topics such as social+media for advertising campaigns (which due to their popularity could involve a high monetary premium), one can focus on peripheral topics, not as popular, but still be able to target an audience close to that of the original query.

Running IAT takes about 0.1 seconds on average for queries evaluated. Note that the majority of the run time to identify analogous topics is taken by the first step. The total run time is bound by the time required to calculate the correlation between the topics.

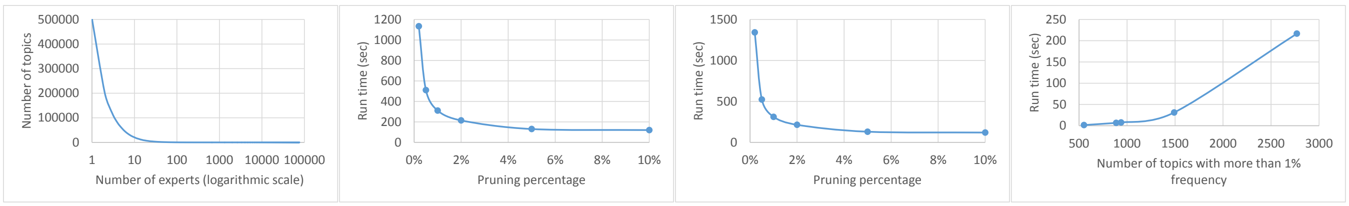
5.2 Categorizing experts and topics

The experts and topics categorization is done in two steps: (1) creating the correlation graph, and (2) executing CTE. As Figure 2(c) shows pruning significantly reduces the run time here as well for query social+media. We report the results when a pruning percentage of 1% is utilized. Similar behavior is observed for the other cases. We define the *size* of a query as the number of topics in $\bigcup_{u \in E_q} T_u$ after pruning takes place. Figure 2(d) reports the run time of CTE versus the size for different queries. On average, CTE takes less than 1 minute to run, making CTE practical for most real settings. As Theorem 5 suggests, the run time of CTE increases polynomially when the size increases.

We have evaluated the CTE algorithm for many queries and observed similar trends in results of all experiments. In what follows, due to space constraints, we present results using the query social+media. We stress however that these results are typical and consistent across a wide range of queries we experimented with. Thus, the specific query social+media is representative of the results obtained with algorithm CTE. Table 3 presents the categories identified by CTE for social+media. In each case, we insert an explanation for the set of topics in each category (in bold). It is evident that the contents of each category are highly related; i.e. from that point of view the results do make sense.

Evaluating the output of CTE qualitatively is challenging. To assess the utility of the results of CTE, we need to compare it with other applicable approaches and most importantly obtain confidence that the categories identified are indeed the correct ones. In the absence of ground truth to objectively compare the CTE approach with other applicable approaches (such as clustering), we resort to develop a base reference set that is manually constructed and compare our results against the base set. Running the CTE algorithm on multiple manually created base sets leads to highly consistent results.

To create these base sets, we choose several topics, categorize each topic into subsets pre-selected by us, and manually annotate each set with a descriptive name. Hereafter, we call these new datasets, the *manually annotated datasets*. These manually annotated datasets, present a "ground truth" in which we know (or expect) a preset number of categories to appear. The goal is to categorize topics and users in the manually annotated datasets utilizing different algorithms (without taking the manual annotations into account) and



(a) topic-expert distribution (b) Identifying Analogous topics (c) Categorizing experts (d) CTE vs. size

Figure 2: Dataset distribution and run time analysis

Table 3: Topic categories for the query "social+media". Rows represent categories including a description followed by the topics in each category.

Tourism in North America (calgari, ottawa, vancouver, toronto, chicago, san fransisco, seattle; hotel, tourism, travel, beer, wine, restaurant, food, ...)
Australia (melbourn, sydney, australia, aussi)
UK (manchester, europe, uk, london)
Sports (tennis, golf, hockey, baseball, nfl, sport, football)
Health (mental heath, health well, pharmacy, healthcare, doctor, medic, psychology, ...)
Education (edu, edtech, learn, university, science, research, academy, ...)
Investments (invest, economia, economy, financ, realest, realtor, real estat)
South by South "SXSW" festivals (sxsw, west texas, austin, houston, dallas)
Law (legal, law, lawyer)
Twibes: groups of people with common interests (twibe socialnetwork, twibe journal, twibe blog, twibe writer, twibe travel, twibe photographi, twibe webdesign, twibe internetmarket, twibe brand, twibe socialmedia, twibe advertis, twibe entrepreneur, ...)

compare how close the results are to the manual annotations. We compare CTE with the baseline clustering algorithm k-means (denoted by kmeans in Figure 3) and 3 baseline co-clustering algorithms Euclidean distance (denoted by cocluster Euclidean), Information theoretic (denoted by cocluster IT), and minimum sum-squared residue co-clustering (denoted by cocluster MSR) [3,6,7]. The following categories form one sample manually annotated dataset:
(1) A category S_1 including topics {physics, math, chemistry}. We call this category, SCIENCE.
(2) A category S_2 including topics {democrats, republican, politics}. We call this category, POLITICS.
(3) A category S_3 including topics {soccer, football, fifa}. We call this category, SPORTS.
(4) A category S_4 including topics {google, tablet, android}. We call this category, TECHNOLOGY.
We create a dataset $\mathcal{D}$. The set of topics in $\mathcal{D}$ is $S =$ {physics, math, chemistry, democrats, republican, politics, soccer, football, fifa, google, tablet, android}. Users in $\mathcal{D}$ are all users who are expert in at least one topic in S ($U' = \{u \in U | T_u \cap S \neq \emptyset\}$ where U denotes the set of all users in the Twitter dataset). For each user $u \in U'$, $T_u \cap S$ is the set of all topics (among topics in $\mathcal{D}$) that u is an expert on. We compare the results of CTE and the baseline algorithms when deployed to categorize users and topics in $\mathcal{D}$.

The optimal categorization of $\mathcal{D}$ is achieved when (1) the topics in S are categorized into 4 categories S_1, S_2, S_3, and S_4 (in accordance with the way the data set was constructed); and (2) the users in U' are categorized into 4 categories of {users who are expert on a topic in S_1}, $\cdots$, {users who are expert on a topic in S_4}.

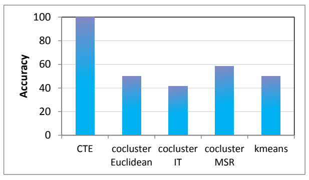
Although algorithm CTE does not need a number of categories as input, the baseline clustering techniques do require the number of clusters (categories). Thus, we provide them with the optimal number 4 providing them with an advantage. Algorithm CTE identifies the optimal number of categories without receiving it as an input.

To calculate the accuracy of an algorithm, we proceed as follows. Assume algorithm X outputs topic categories $C_1, C_2, \cdots, C_r$ and user categories $D_1, D_2, \cdots, D_s$. We utilize four annotations SCIENCE, POLITICS, SPORTS, and TECHNOLOGY to label each category produced by X. A category C_i (D_i) is labeled by the annotation having the maximum number of entities in that category. For example, consider a topic category $C_1 =$ {physics, soccer, math}. This category includes two topics in SCIENCE, one topic in SPORTS, and no topic in POLITICS or TECHNOLOGY. Thus, we label the category C_1 as SCIENCE. Moreover, assume $D_1 = \{u_1, u_2, u_3, u_4\}$, where $T_{u_1} =$ {soccer, math}, $T_{u_2} =$ {soccer}, $T_{u_3} =$ {math, democrats}, and $T_{u_4} =$ {chemistry, republican}. We can observe that 3 users in D_1 are experts on SCIENCE, 2 users on SPORTS, and 2 users on POLITICS. Therefore, we label the category D_1 as SCIENCE.

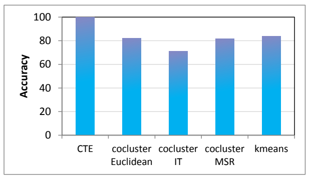
The topic categorization accuracy (user categorization accuracy) of an algorithm is the percentage of the topics (users) that are labeled correctly. Note that in the previous example, one topic is labeled inaccurately in C_1 (the topic soccer is labeled as SCIENCE) and one user is labeled inaccurately in D_1 (u_2 is labeled as SCIENCE without having any expertise on physics, math, or chemistry). Figure 3(a) reports the accuracy for topic categories and Figure 3(b) shows the accuracy for user categories for all algorithms. Figure 3 demonstrates the superiority of CTE when compared with baseline clustering algorithms.

6. RELATED WORKS

Twitter lists are recently used to address a few questions such as identifying users' topics of expertise [2,16] and separating elite users (e.g., celebrities) from ordinary users [18]. The problem of identifying a set of topics that can be utilized as a substitute for an expensive topic is studied for the case that target sets of topics are given and the cost for each topic is known [9]. In many real settings we don't have ac-



(a) Topic categories



(b) User categories

Figure 3: Comparison between the accuracy of different clustering algorithms

cess to this information. This paper focuses on the problem when the target sets and costs are unknown.

Automatons are utilized in several problems such as identifying bursts of activity in time-series data [13], spatial datasets [14], and subgraphs of social networks' graphs [8]. Perhaps the most similar work to our IAT algorithm is the DIBA algorithm [8] that is proposed to identify the bursty subgraphs of users in a social network when the information burst happens as a result of an external activity (such as an earthquake). We note that there are major differences between our IAT and DIBA algorithms: (1) DIBA is mainly designed for unweighted graphs; (2) DIBA does not consider negative edges. In fact, the optimization problem (Problem 2) in presence of negative edges is NP-hard (Theorem 3) while if all weights are non-negative, the problem would become equivalent to min-cut and can be solved in polynomial time [8]; (3) IAT addresses Problem 2 by locating the optimal cycle-free subgraph, while DIBA utilizes a heuristic approach that randomly orders graph nodes and attempts to find the best label for each node in this order; this approach does not identify the optimal subgraph and may ignore considering several important (costly) edges.

The traditional clustering algorithms can be categorized to partitioning methods (e.g., k-means), hierarchical methods (top-down, bottom-up), Density-based (e.g., DBSCAN), model-based (EM), link-based, bi-clustering, and graph partitioning (e.g., finding cliques or quasi-cliques in the graph, and correlation clustering). These algorithms also suffer from several disadvantages in the case of our problem. To the best of our knowledge none of these clustering algorithms provide all of the four desirable properties (introduced in Section 4.1); hence they are not applicable to categorize experts. For completeness, we compared our proposed algorithm with some of these algorithms in Section 5.

7. CONCLUSION AND FUTURE WORKS

In this paper we introduce two problems. The first problem is to identify topics (called analogous) that have (approximately) the same audience on a micro-blogging platforms as a query topic. The idea is that by bidding on an analogous topic instead of the original query topic, we will reach (approximately) the same audience while spending less budget on advertising. This is inspired by the social media advertising platforms. The second problem is to understand the diversified expertise of the experts on the given query topic and categorize these experts. We evaluate the techniques proposed for both problems on a large dataset from Twitter attesting their efficiency and accuracy.

An important direction for future work is to study the problems when the bids on each topic is known. This extension can assist advertisers to maximize their revenue while minimizing the advertising cost.

8. ACKNOWLEDGMENTS

The authors would like to thank Alex Cheng for his assistance with data preparation.

9. REFERENCES

[1] N. Bansal, A. Blum, and S. Chawla. Correlation clustering. *Mach. Learn.*, 56(1-3):89–113, June 2004.

[2] A. Cheng, N. Bansal, and N. Koudas. Peckalytics: Analyzing experts and interests on twitter. SIGMOD Demo Track, 2013.

[3] H. Cho, I. Dhillon, Y. Guan, and S. Sra. Minimum sum-squared residue co-clustering of gene expression data. SDM, pages 114–125, 2004.

[4] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 3rd edition, 2009. Chapter 23.

[5] E. D. Demaine, D. Emanuel, A. Fiat, and N. Immorlica. Correlation clustering in general weighted graphs. *Theor. Comput. Sci.*, 361(2):172–187, Sept. 2006.

[6] I. Dhillon and Y. Guan. Information theoretic clustering of sparse co-occurrence data. ICDM, pages 517–520, 2003.

[7] I. S. Dhillon, S. Mallela, and D. S. Modha. Information theoretic co-clustering. SIGKDD, pages 89–98, 2003.

[8] M. Eftekhar, N. Koudas, and Y. Ganjali. Bursty subgraphs in social networks. WSDM, pages 213–222, 2013.

[9] M. Eftekhar, S. Thirumuruganathan, G. Das, and N. Koudas. Price trade-offs in social media advertising. In *Proceedings of the Second Edition of the ACM Conference on Online Social Networks*, COSN '14, pages 169–176, New York, NY, USA, 2014. ACM.

[10] Facebook. Facebook help centre. `https://www.facebook.com/help/ads`.

[11] J. Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, June 1956.

[12] R. Karp. Reducibility among combinatorial problems. In *50 Years of Integer Programming 1958-2008*, pages 219–241. Springer Berlin Heidelberg, 2010.

[13] J. Kleinberg. Bursty and hierarchical structure in streams. SIGKDD, pages 91–101, 2002.

[14] M. Mathioudakis, N. Bansal, and N. Koudas. Identifying, attributing and describing spatial bursts. *VLDB Endowment*, 3(1-2):1091–1102, Sept. 2010.

[15] J. L. Rodgers and A. W. Nicewander. Thirteen ways to look at the correlation coefficient. *The American Statistician*, 42(1):59–66, 1988.

[16] N. K. Sharma, S. Ghosh, F. Benevenuto, N. Ganguly, and K. Gummadi. Inferring who-is-who in the twitter social network. In *Proceedings of the 2012 ACM workshop on Workshop on online social networks*, pages 55–60, 2012.

[17] Twitter. Start Advertising | Twitter for Business. `https://business.twitter.com/start-advertising`.

[18] S. Wu, J. M. Hofman, W. A. Mason, and D. J. Watts. Who says what to whom on twitter. In *Proceedings of the 20th international conference on World wide web*, pages 705–714, 2011.

On Optimality of Jury Selection in Crowdsourcing

Yudian Zheng, Reynold Cheng, Silviu Maniu, and Luyi Mo
Department of Computer Science, University of Hong Kong, Pokfulam Road, Hong Kong
{ydzheng2, ckcheng, smaniu, lymo}@cs.hku.hk

ABSTRACT

Recent advances in crowdsourcing technologies enable computationally challenging tasks (e.g., sentiment analysis and entity resolution) to be performed by Internet workers, driven mainly by monetary incentives. A fundamental question is: how should workers be selected, so that the tasks in hand can be accomplished successfully and economically? In this paper, we study the *Jury Selection Problem* (JSP): Given a monetary budget, and a set of decision-making tasks (e.g., "Is Bill Gates still the CEO of Microsoft now?"), return the set of workers (called *jury*), such that their answers yield the highest "Jury Quality" (or JQ). Existing JSP solutions make use of the *Majority Voting* (MV) strategy, which uses the answer chosen by the largest number of workers. We show that MV does not yield the best solution for JSP. We further prove that among all voting strategies (including deterministic and randomized strategies), *Bayesian Voting* (BV) can optimally solve JSP. We then examine how to solve JSP based on BV. This is technically challenging, since computing the JQ with BV is NP-hard. We solve this problem by proposing an approximate algorithm that is computationally efficient. Our approximate JQ computation algorithm is also highly accurate, and its error is proved to be bounded within 1%. We extend our solution by considering the task owner's "belief" (or *prior*) on the answers of the tasks. Experiments on synthetic and real datasets show that our new approach is consistently better than the best JSP solution known.

1. INTRODUCTION

Due to advances in crowdsourcing technologies, computationally challenging tasks (e.g., sentiment analysis, entity resolution, document translation, etc.) can now be easily performed by human workers on the Internet. As reported by the Amazon Mechanical Turk in August 2012, over 500,000 workers from 190 countries worked on human intelligence tasks (HITs). The large number of workers and HITs have motivated researchers to develop solutions to streamline the crowdsourcing process [6,7,14,25,27,31,43,44].

In general, crowdsourcing a set of tasks involves the following steps: (1) distributing tasks to workers; (2) collecting the workers' answers; (3) deciding final result; and (4) rewarding the workers.

An important question is: how should workers be chosen, so that the tasks in hand can be completed with high quality, while minimizing the monetary budget available? A related question, called the *Jury Selection Problem* (or JSP), has been recently proposed by Cao et al. [7]. Similar to the concept from law courts, a *jury*, or *jury set* denotes a subset of workers chosen from the available worker pool. Given a monetary budget and a task, the goal of JSP is to find the jury with the highest expected performance within the budget constraint. The kind of tasks studied in [7] is called the *decision-making task*: a question that requires an answer of either *yes* or *no* (e.g., "Is Bill Gates still the CEO of Microsoft now?") and has a definitive ground truth. Decision-making tasks [7,39,41,44] are commonly used in crowdsourcing systems because of their conceptual simplicity. The authors of [7] were the first to propose a system to address JSP for this kind of tasks.

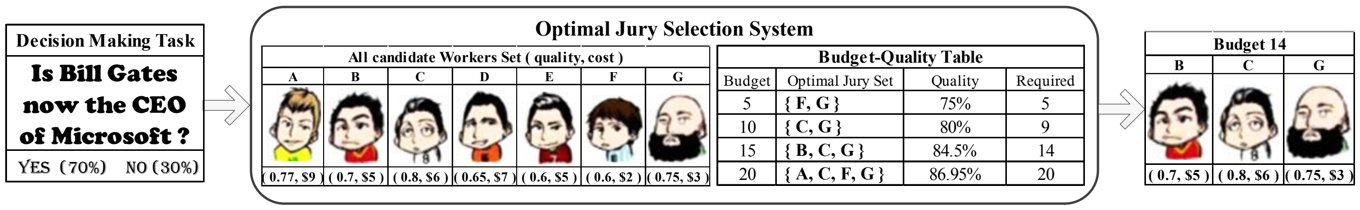
In this paper, we go beyond [7] and perform a comprehensive investigation of this problem. Particularly, we ask the following questions: (1) Is the solution in [7] optimal? (2) If not, what is an optimal solution for JSP? To understand these issues, let us first illustrate how [7] solves JSP.

Figure 1 shows a decision-making task, to be answered by some of the seven workers labeled from A to G where each worker is associated with a *quality* and a *cost*. The *quality* ranges from 0 to 1, indicating the probability that the worker correctly answers a question. This probability can be estimated by using her background information (e.g., her performance in other tasks) [7,25,37]. The *cost* is the amount of monetary reward the worker can get upon finishing a task. In this example, A has a quality of 0.77 and a cost of 9 units. For a jury, the *jury cost* is defined as the sum of workers' costs in the jury and the *jury quality* (or JQ) is defined as the probability that the result returned by aggregating the jury answers is correct. Given a budget of B units, a feasible jury is a jury whose *jury cost* does not exceed B. For example, if $B = \$20$, then $\{B, E, F\}$ is a feasible jury, since its *jury cost*, or $\$5 + \$5 + \$2 = \$12$, is not larger than \$20.

To solve JSP, a naive solution is to compute the JQ for every feasible jury, and return the one with the highest JQ. [7] studies how to compute JQ for a jury where the jury's returned result is decided by *Majority Voting* (MV). In short, MV returns the result as the one corresponding to the most workers. In the following, we consider each worker's answer as a "vote" for either "yes" or "no". Let us consider $\{B, E, F\}$ again, the probability that these workers gives a correct result according to MV is $0.7 \cdot 0.6 \cdot 0.6 + 0.7 \cdot 0.6 \cdot (1 - 0.6) + 0.7 \cdot (1 - 0.6) \cdot 0.6 + (1 - 0.7) \cdot 0.6 \cdot 0.6 = 69.6\%$. Moreover, since $\{A, C, G\}$ yields the highest JQ among all the feasible juries, it is considered to be the best solution by [7].

As illustrated above, MV is used to solve JSP in [7]. In addition to MV, researchers have proposed a variety of *voting strategies*,

10.5441/002/edbt.2015.18



Budget-Quality Table

Budget	Optimal Jury Set	Quality	Required
5	**{ F, G }**	75%	5
10	**{ C, G }**	80%	9
15	**{ B, C, G }**	84.5%	14
20	**{ A, C, F, G }**	86.95%	20

Figure 1: Optimal Jury Selection System.

such as Bayesian Voting (BV) [25], Randomized Majority Voting [20], and Random Ballot Voting [33]. Like MV, these voting strategies decide the final result of a decision-making task based on the workers' votes. For example, BV computes the posterior probability of answers according to Bayes' Theorem [3], based on the workers' votes, and returns the answer having the largest posterior probability.

In this paper, we investigate an interesting problem: is it possible to find the optimal voting strategy for JSP among all voting strategies? One simple answer to this question is to consider all voting strategies. However, as listed in Table 2, the number of existing strategies is very large. Moreover, multiple new strategies may emerge in the future. We address this question by first studying the criteria of a strategy that produce an optimal solution for JSP (i.e., given a jury, the JQ of the strategy is the highest among all the possible voting strategies). This is done by observing that voting strategies can be classified into two major categories: *deterministic* and *randomized*. A deterministic strategy aggregates workers' answers without any degree of randomness; MV is a typical example of this class. For a randomized strategy, each answer is returned with some probability. Using this classification, we present the criteria required for a voting strategy that leads to the optimal solution for JSP. We discover that BV satisfies the requirements of an optimal strategy. In other words, BV is the optimal voting strategy with respect to JQ, and will consistently produce better quality juries than the other strategies.

How to solve JSP with BV then? A straightforward solution is to enumerate all feasible juries, and find the one with the largest value of JQ. However, this approach suffers from two major problems:

1. Computing the JQ of a jury for BV requires enumerating an exponentially large number of workers' answers. In fact, we show that this problem is NP-hard;
2. The number of feasible juries is exponentially large.

To solve Problem 1, we develop a polynomial-time-based approximation algorithm, which enables a large number of candidate answers to be pruned, without a significant loss of accuracy. We further develop a theoretical error bound of this algorithm. Particularly, our approximate JQ computation algorithm is proved to yield an error of not more than 1%. To tackle Problem 2, we leverage a successful heuristic, the simulated annealing heuristic, by designing local neighborhood search functions. To evaluate our solutions, we have performed extensive evaluation on real and synthetic crowdsourced data. Our experimental results show that our algorithms effectively and efficiently solve JSP. The quality of our solution is also consistently better than that of [7].

We also study how to allow the provider of the tasks to place her confidence information (called *prior*) on the answers of the task. She may associate a "belief score" on the answers to the tasks, before the crowdsourcing process starts. For instance, in Figure 1, if she is more confident that Bill Gates is still the CEO of Microsoft, she can assign 70% to *yes*, and 30% to *no*. Intuitively, we prove that under BV, the effect of prior is just the same as regarding the task provider as another worker, having the same quality values as the prior.

Figure 1 illustrates our crowdsourcing system, which we called the "Optimal Jury Selection System". In this system, the task provider published a decision-making task. Then, based on the the workers' information (i.e., their individual quality and cost), a "budget-quality table" is generated. In this table, each row contains a budget, the computed optimal jury, its estimated jury quality and the required budget for the jury. Based on this table, the task provider can conveniently decide the best budget-quality combination. For example, she may deem that increasing the budget from 15 units to 20 units is not worthwhile, since the quality increases only by around 2.5%. In this example, the task provider selects the jury set $\{B, C, G\}$ that is the best under a budget of 15 units. This chosen jury set would cost her only 14 units.

Recall that [7] focuses on addressing JSP under MV on decision-making tasks and we address the optimality of JSP on decision-making tasks by considering all voting strategies, where each worker's quality is modeled as a single parameter. In reality, multiple choice tasks [25,34,42] are also commonly used in crowdsourcing and several works [1,34,36] model each worker as a confusion matrix rather than a single quality score. We also briefly discuss here the optimality of JSP for other task types and worker models, and how our solutions can be extended to these other variants.

The rest of this paper is arranged as follows. We describe the data model and the problem definition in Section 2. In Section 3, we examine the requirements of an optimal voting strategy for JSP, and show that BV satisfies these criteria. We present an efficient algorithm to compute JQ of a jury set in Section 4 and develop fast solutions to solve JSP in Section 5. In Section 6, we present our experimental results. We discuss how our solutions can be extended for other task types and worker models in Section 7. In Section 8, we review the related works and Section 9 concludes the paper.

2. DATA MODEL & PROBLEM DEFINITION

We now describe our data model in Section 2.1 and define the jury selection problem in Section 2.2.

2.1 Data Model

In this paper, we focus on the *decision-making tasks* where each task has two possible answers (either *yes* or *no*). We use 1 and 0 to denote *yes* and *no*, respectively. We assume that each task has a latent true answer (or ground truth) $\mathbf{t} \in \{0, 1\}$, which is unknown in advance. The task provider usually assigns a *prior* on the task, which describes her prior knowledge in the probability distribution of the task's true answer. We denote the prior by α where $\Pr(\mathbf{t} = 0) = \alpha$, and $\Pr(\mathbf{t} = 1) = 1-\alpha$. If the task provider has no prior knowledge for the task, then we assume $\alpha = 0.5$.

A *jury* (or *jury set*), denoted by J, is a collection of n workers drawn from a set of N candidate workers $W = \{j_1, j_2, \ldots, j_N\}$, i.e., $J \subseteq W$, $|J| = n$. Without loss of generality, let $J =$

$\{j_1, j_2, \ldots, j_n\}$. In order to infer the ground truth ($\mathbf{t}$), we leverage the collective intelligence of a jury, i.e, we ask each worker to give a vote for the task. We use V, a *voting*, to denote the set of votes (answers) given by a jury J, and so $V = \{v_1, v_2, \ldots, v_n\}$ where $v_i \in \{0, 1\}$ is the vote given by j_i. We assume the independence of each worker's vote, an assumption also used in [7,18,25,34].

We follow the worker model in previous works [7,25,44], where each worker j_i is associated with a quality $q_i \in [0, 1]$ and a cost c_i. The quality q_i indicates the probability that the worker conducts a correct vote, i.e., $q_i = \Pr(v_i = \mathbf{t})$, and the cost c_i represents the money (or incentive) required for j_i to give a vote. A few works [7,25,37] have recently addressed how to derive the quality and the cost of a worker by leveraging the backgrounds and answering history of individuals. Thus, similar to [7], we assume that they are known in advance.

We remark that the optimality of JSP and our solutions can be extended to address other task types and worker models used in [1,25,34,34,36,42]. We will briefly discuss these extensions in Section 7.

2.2 Problem Definition

Let B be the budget of a task provider, i.e., a maximum of B cost units can be given to a jury to collect their votes. Our goal is to solve the *Jury Selection Problem* (denoted by JSP) which selects a jury J under the budget constraint ($\sum_{j_i \in J} c_i \leq B$) such that the jury's collective intelligence is maximized.

The collective intelligence of a jury is closely related to the *Voting Strategy*, denoted by S, which estimates the true answer of the task based on the prior, the jury and their votes. We say the estimated true answer is the *result* of the voting strategy. A detailed discussion about the voting strategy is given in Section 3.1.

In order to quantify the jury's collective intelligence, we define the *Jury Quality* (or *JQ* in short) which essentially measures the probability that the result of the voting strategy is correct. The score of JQ is given by function $JQ(J, S, \alpha)$. We will give a precise definition for JQ in Section 3.2.

Let Θ denote the set of all voting strategies and $\mathcal{C}$ denote the set of all feasible juries (i.e., $\mathcal{C} = \{J \mid J \subseteq W \wedge \sum_{j_i \in J} c_i \leq B\}$). The aim of JSP is to select the optimal jury J^* such that

$$\textbf{given} \quad \alpha \text{ and } q_i, c_i \text{ (for } i = 1, 2, \ldots, N) \tag{1}$$

$$J^* = \arg\max_{J \in \mathcal{C}} \max_{S \in \Theta} JQ(J, S, \alpha) \tag{2}$$

Note that existing work [7] only focuses on majority voting strategy (MV) and solves $\arg\max_{J \in \mathcal{C}} JQ(J, MV, 0.5)$, which, as we shall prove later, is sub-optimal for JSP.

In the rest of the paper, we first discuss how to derive the optimal voting strategy S^* such that $JQ(J, S^*, \alpha) = \max_{S \in \Theta} JQ(J, S, \alpha)$ (Section 3). We then talk about the computation of $JQ(J, S^*, \alpha)$ (Section 4) and finally address of problem of finding J^* (Section 5).

Table 1 summarizes the symbols used in this paper.

3. OPTIMAL VOTING STRATEGY

In this section, we present a detailed description for the voting strategy in Section 3.1. We then formally define JQ in Section 3.2. Finally, we give an optimal voting strategy with respect to JQ in Section 3.3.

3.1 Voting Strategies

As mentioned, a *voting strategy* S gives an estimation of the true answer $\mathbf{t}$ based on the prior α, the jury J and their votes V. Thus, we model a voting strategy as a function $S(V, J, \alpha)$, whose result is an estimation of $\mathbf{t}$. Based on whether the result is given with degree of randomness, we can classify the voting strategies into two categories: *deterministic voting strategy* and *randomized voting strategy*.

Table 1: Table of Symbols

Symbol	Description
$\mathbf{t}$	the ground truth for a task, and $\mathbf{t} \in \{0, 1\}$
α	prior given by the task provider, and $\alpha = \Pr(\mathbf{t} = 0)$
W	a set of all candidate workers $W = \{j_1, j_2, \ldots, j_N\}$
J	a jury, $J \subseteq W$ and $\lvert J \rvert = n$, $J = \{j_1, j_2, \ldots, j_n\}$
V	a voting given by J, and $V = \{v_1, v_2, \ldots, v_n\}$
q_i	the quality of worker j_i and $q_i \in [0, 1]$
c_i	the cost of worker j_i
B	the budget provided by the task provider
Θ	a set containing all voting strategies
$\mathcal{C}$	the set of all possible juries within budget constraint

DEFINITION 1. *A deterministic voting strategy $S(V, J, \alpha)$ returns the result as 0 or 1 without any degree of randomness.*

DEFINITION 2. *A randomized voting strategy $S(V, J, \alpha)$ returns the result as 0 with probability p and 1 with probability $1 - p$.*

EXAMPLE 1. *The majority voting strategy (or MV) is a typical deterministic voting strategy, and it gives result as* 0 *if more than half of workers vote for* 0 *(i.e., $\sum_{i=1}^{n} (1 - v_i) \geq \frac{n+1}{2}$); otherwise, the result is* 1.

Its randomized counterpart is called randomized majority voting strategy (or RMV), which returns the result with probability proportional to the number of votes. That is, RMV returns 0 *with probability $p = \frac{1}{n} \sum_{i=1}^{n} (1 - v_i)$, and* 1 *with probability $1 - p$.*

Note that randomized strategies are often introduced to improve the error bound for worst-case analysis [23]. And thus, they are widely used when the worst-case performance is the main concern.

Table 2: Classification of Voting Strategies

Deterministic Voting Strategies	Randomized Voting Strategies
Majority Voting (MV) [7]	Randomized Majority Voting (RMV) [20]
Half Voting [28]	Random Ballot Voting [33]
Bayesian Voting [25]	Triadic Consensus [2]
Weighted MV [23]	Randomized Weighted MV [23]
...	...

Table 2 shows a few voting strategies, which are introduced in previous works, and their corresponding category.

3.2 Jury Quality

In order to measure the goodness of a voting strategy S for a jury J, we introduce a metric called *Jury Quality* (or *JQ* in short). We model JQ by a function $JQ(J, S, \alpha)$ which gives the quality score as the probability of drawing a correct result under the voting strategy, i.e.,

$$JQ(J, S, \alpha) = \Pr(S(\mathbf{V}, J, \alpha) = \mathbf{t}) \tag{3}$$

where $\mathbf{V} \in \{0, 1\}^n$ and $\mathbf{t} \in \{0, 1\}$ are two random variables corresponding to the unknown jury's voting, and the task's latent true answer. For notational convenience, we omit J and α in S when their values are understood and simply write $S(\mathbf{V})$ instead of $S(\mathbf{V}, J, \alpha)$.

Let $\mathbb{1}_{\{st\}}$ be the indicator function, which returns 1 if the statement st is true, and 0 otherwise. Let Ω be the domain of $\mathbf{V}$, i.e,

$\Omega = \{0,1\}^n$. $JQ(J, S, \alpha)$ can be rewritten as follows.

$$\begin{aligned} JQ(J,S,\alpha) &= 1 \cdot \Pr(S(\mathbf{V}) = \mathbf{t}) + 0 \cdot \Pr(S(\mathbf{V}) \neq \mathbf{t}) \\ &= \mathbb{E}[\mathbb{1}_{\{S(\mathbf{V})=\mathbf{t}\}}] \\ &= \sum_{t \in \{0,1\}} \sum_{V \in \Omega} \Pr(\mathbf{V} = V, \mathbf{t} = t) \cdot \mathbb{E}[\mathbb{1}_{\{S(V)=t\}}] \end{aligned}$$

We now give a precise definition for JQ as below.

DEFINITION 3 (JURY QUALITY). *Given a jury J and the prior α, the Jury Quality (or JQ) for a voting strategy S, denoted by $JQ(J, S, \alpha)$, is defined as*

$$\begin{aligned} &\alpha \cdot \sum\nolimits_{V \in \Omega} \Pr(\mathbf{V} = V \mid \mathbf{t} = 0) \cdot \mathbb{E}[\mathbb{1}_{\{S(V)=0\}}] \\ &+ (1-\alpha) \cdot \sum\nolimits_{V \in \Omega} \Pr(\mathbf{V} = V \mid \mathbf{t} = 1) \cdot \mathbb{E}[\mathbb{1}_{\{S(V)=1\}}]. \end{aligned} \tag{4}$$

For notational convenience, we write $\Pr(V|\mathbf{t} = 0)$ instead of $\Pr(\mathbf{V} = V|\mathbf{t} = 0)$, and $\Pr(V|\mathbf{t} = 1)$ instead of $\Pr(\mathbf{V} = V|\mathbf{t} = 1)$. Next, we give two marks in computing JQ.

1. Since workers give votes independently, we have

$$\begin{aligned} \Pr(V \mid \mathbf{t} = 0) &= \prod\nolimits_{i=1}^{n} q_i^{(1-v_i)} \cdot (1-q_i)^{v_i} \\ \Pr(V \mid \mathbf{t} = 1) &= \prod\nolimits_{i=1}^{n} q_i^{v_i} \cdot (1-q_i)^{(1-v_i)} \end{aligned}$$

2. $\mathbb{E}[\mathbb{1}_{\{S(V)=0\}}]$ and $\mathbb{E}[\mathbb{1}_{\{S(V)=1\}}]$ are either 0 or 1 if S is a deterministic voting strategy; or value of p and $1 - p$ if S is a randomized voting strategy (refer to Definition 2).

We next give an example to illustrate the computation of JQ.

EXAMPLE 2. *Suppose $\alpha = 0.5$ and there are 3 workers in J with workers' qualities as $0.9, 0.6, 0.6$ respectively. To compute JQ for MV, we enumerate all possible combinations of V $(\in \{0,1\}^3)$ and t $(\in \{0,1\})$, and show the results in Figure 2. The 3rd column in each table represents the probability that a specific combination (V and t) exists. The 4th column shows the result of MV for each V. The symbol $\surd$ indicates whether MV's result is correct or not (according to the value of t). And thus, $JQ(J, MV, \alpha)$ equals to the summation of probabilities where symbol $\surd$ occurs. Take $V = \{1,0,0\}$ and $t = 0$ as an example. First, $\Pr(\mathbf{V} = V, \mathbf{t} = 0) = 0.018$. Since $\sum_{i=1}^{3}(1 - v_i) = 2 \geq \frac{n+1}{2} = 2$, we have $MV(V) = 0 = t$. Thus, the probability 0.018 is added to $JQ(J, MV, \alpha)$. Similarly, for $V = \{1,0,0\}$ and $t = 1$, as $MV(V) = 0 \neq t$, then $\Pr(\mathbf{V} = V, \mathbf{t} = 1) = 0.072$ will not be added to $JQ(J, MV, \alpha)$. Considering all V's and t's, the final $JQ(J, MV, \alpha) = 79.2\%$.*

3.3 Optimal Voting Strategy

In the last two sections, we present a few voting strategies and define Jury Quality to quantify the goodness of a voting strategy. Thus an interesting question is: does an optimal voting strategy S^* with respect to JQ exist? That is, given any J and α, $JQ(J, S^*, \alpha) = \max_{S \in \Theta} JQ(J, S, \alpha)$. Note that if S^* exists, we can then solve JSP without enumerating all voting strategies in Θ (refer to Equation 2).

To answer this question, let us reconsider Definition 3. Let $h(V) = \mathbb{E}[\mathbb{1}_{\{S(V)=0\}}]$. We have (i) $h(V) \in [0,1]$; and (ii) $\mathbb{E}[\mathbb{1}_{\{S(V)=1\}}] = 1 - h(V)$. Also, let $P_0(V) = \Pr(\mathbf{V} = V, \mathbf{t} = 0)$, and $P_1(V) = \Pr(\mathbf{V} = V, \mathbf{t} = 1)$. Hence, $JQ(J, S, \alpha)$ can be rewritten as

$$\begin{aligned} &\sum\nolimits_{V \in \Omega} [\, P_0(V) \cdot h(V) + P_1(V) \cdot (1 - h(V)) \,] \\ = &\sum\nolimits_{V \in \Omega} [\, h(V) \cdot (P_0(V) - P_1(V)) + P_1(V) \,] \end{aligned}$$

No.	V	P(t = 0)*P(**V** = *V* | **t** = 0)	MV: [compare] (result) [√/×]	BV: [compare] (result) [√/×]
1	{ 0,0,0 }	0.5*0.9*0.6*0.6=0.162	[3 ≥ 2] (0) [√]	[0.162 ≥ 0.008] (0) [√]
2	{ 0,0,1 }	0.5*0.9*0.6*0.4=0.108	[2 ≥ 2] (0) [√]	[0.108 ≥ 0.012] (0) [√]
3	{ 0,1,0 }	0.5*0.9*0.4*0.6=0.108	[2 ≥ 2] (0) [√]	[0.108 ≥ 0.012] (0) [√]
4	{ 0,1,1 }	0.5*0.9*0.4*0.4=0.072	[1 < 2] (1) [×]	[0.072 ≥ 0.018] (0) [√]
5	{ 1,0,0 }	0.5*0.1*0.6*0.6=0.018	[2 ≥ 2] (0) [√]	[0.018 < 0.072] (1) [×]
6	{ 1,0,1 }	0.5*0.1*0.6*0.4=0.012	[1 < 2] (1) [×]	[0.012 < 0.018] (1) [×]
7	{ 1,1,0 }	0.5*0.1*0.4*0.6=0.012	[1 < 2] (1) [×]	[0.012 < 0.018] (1) [×]
8	{ 1,1,1 }	0.5*0.1*0.4*0.4=0.008	[0 < 2] (1) [×]	[0.008 < 0.162] (1) [×]

(a) Enumeration of all $2^3 = 8$ possible votings in Ω ($\mathbf{t} = 0$)

No.	V	P(t = 1)*P(**V** = *V* | **t** = 1)	MV: [compare] (result) [√/×]	BV: [compare] (result) [√/×]
1	{ 0,0,0 }	0.5*0.1*0.4*0.4=0.008	[3 ≥ 2] (0) [×]	[0.162 ≥ 0.008] (0) [×]
2	{ 0,0,1 }	0.5*0.1*0.4*0.6=0.012	[2 ≥ 2] (0) [×]	[0.108 ≥ 0.012] (0) [×]
3	{ 0,1,0 }	0.5*0.1*0.6*0.4=0.012	[2 ≥ 2] (0) [×]	[0.108 ≥ 0.012] (0) [×]
4	{ 0,1,1 }	0.5*0.1*0.6*0.6=0.018	[1 < 2] (1) [√]	[0.072 ≥ 0.018] (0) [×]
5	{ 1,0,0 }	0.5*0.9*0.4*0.4=0.072	[2 ≥ 2] (0) [×]	[0.018 < 0.072] (1) [√]
6	{ 1,0,1 }	0.5*0.9*0.4*0.6=0.108	[1 < 2] (1) [√]	[0.012 < 0.018] (1) [√]
7	{ 1,1,0 }	0.5*0.9*0.6*0.4=0.108	[1 < 2] (1) [√]	[0.012 < 0.018] (1) [√]
8	{ 1,1,1 }	0.5*0.9*0.6*0.6=0.162	[0 < 2] (1) [√]	[0.008 < 0.162] (1) [√]

(b) Enumeration of all $2^3 = 8$ possible votings in Ω ($\mathbf{t} = 1$)

Figure 2: Example of JQ computation for MV and BV ($\alpha = 0.5$, and the quality of workers are $0.9, 0.6$, and 0.6)

This gives us a hint to maximize $JQ(J, S, \alpha)$ and find the optimal voting strategy S^*. Let $h^*(V) = \mathbb{E}[\mathbb{1}_{\{S^*(V)=0\}}]$. It is observed that $P_1(V)$ is constant for a given V and $h(V) \in [0,1]$ for all S's (no matter it is a deterministic one or a randomized one). Thus, to optimize $JQ(J, S, \alpha)$, it is required that

1. if $P_0(V) - P_1(V) < 0$, $h^*(V) = 0$, and so, $S^*(V) = 1$;
2. if $P_0(V) - P_1(V) \geq 0$, $h^*(V) = 1$, and so, $S^*(V) = 0$.

We summarize this observation as below.

THEOREM 1. *Given α, J, and V, the optimal voting strategy, denoted by S^*, decides the result as follows:*

1. *$S^*(V) = 1$ if $\alpha \cdot \prod_{i=1}^{n} q_i^{(1-v_i)} \cdot (1-q_i)^{v_i} <$ $(1-\alpha) \cdot \prod_{i=1}^{n} q_i^{v_i} \cdot (1-q_i)^{(1-v_i)}$; or*
2. *$S^*(V) = 0$, otherwise.*

Note that S^* is a deterministic voting strategy, and it's essentially a voting strategy based on the Bayes' Theorem [11]. The reason is as follows. According to the Bayes' Theorem, based on the observed voting V, $\Pr(\mathbf{t} = 0|\mathbf{V} = V) = P_0(V)/\Pr(\mathbf{V} = V)$, and similarly $\Pr(\mathbf{t} = 1|\mathbf{V} = V) = P_1(V)/\Pr(\mathbf{V} = V)$. Therefore, $P_0(V) - P_1(V) < 0$ indicates $\Pr(\mathbf{t} = 0|\mathbf{V} = V) < \Pr(\mathbf{t} = 1|\mathbf{V} = V)$. And so, 1 has a higher probability to be the true answer than 0. Thus, the voting strategy based on the Bayes' Theorem returns 1 as the result, which is consistent with S^* in Theorem 1. Next, we give a formal definition for Bayesian Voting (BV) and summarize the above observation in Theorem 1.

DEFINITION 4. *The voting strategy based on the Bayes' Theorem, denoted by Bayesian Voting (or BV in short), returns the result as 1, if $\Pr(\mathbf{t} = 0) \cdot \Pr(\mathbf{V} = V|\mathbf{t} = 0) < \Pr(\mathbf{t} = 1) \cdot \Pr(\mathbf{V} = V|\mathbf{t} = 0)$; or 0, otherwise.*

COROLLARY 1. *BV is optimal w.r.t. JQ, i.e., $S^* = BV$.*

Note that the BV is also used in [1,18,25]. In the rest of the paper, we use S^* and BV interchangeably. We remark that the optimality of BV is based on two assumptions: (1) the prior and workers' qualities are known in advance; (2) JQ (Definition 3) is adopted to measure the goodness of a voting strategy.

EXAMPLE 3. *Let us reconsider Figure 2 and see how $JQ(J, BV, \alpha)$ is computed. The 5th column shows results given by BV. The two numbers in bracket correspond to $P_0(V)$ and $P_1(V)$, respectively. The value in parenthesis is the estimated true answer returned by BV. We again use a symbol $\surd$ to indicate the correct voting result. Take $V = \{1, 0, 0\}$ and $t = 0$ as an example. Since $\alpha \cdot (1 - q_1) \cdot q_2 \cdot q_3 = 0.018 < (1 - \alpha) \cdot q_1 \cdot (1 - q_2) \cdot (1 - q_3) = 0.072$, we have $BV(V) = 1 \neq t$, thus 0.018 is not added into $JQ(J, BV, \alpha)$. Otherwise, for $V = \{1, 0, 0\}$ and $t = 0$, similarly we derive that 0.072 is added in $JQ(J, BV, \alpha)$. Recall Example 2, when $V = \{1, 0, 0\}$, if we consider two cases of t, then 0.072 is added into $JQ(J, BV, \alpha)$; but here we have seen in Example 2 that 0.018 is added into $JQ(J, MV, \alpha)$. By considering all V and t, we have $JQ(J, BV, \alpha) = 90\% > JQ(J, MV, \alpha) = 79.2\%$.*

Intuitively, the reason why BV outperforms other voting strategies is that BV considers the prior and worker's qualities in deriving the result of a voting V, and only the one with larger posterior probability is returned. Thus, it is more likely to return a correct answer than other strategies. For example, assume $\alpha = 0.5$ and the voting $V = \{0, 1, 1\}$ is given by workers with individual quality 0.9, 0.6 and 0.6 respectively. As $0.5 \cdot 0.9 \cdot (1 - 0.6) \cdot (1 - 0.6) > 0.5 \cdot (1 - 0.9) \cdot 0.6 \cdot 0.6$, BV returns 0 as the result. However, MV does not leverage either the prior information or workers' qualities, and so, it returns 1, which is given by two lower quality workers.

Before we move on, we would like to discuss the effect of q_i for voting strategies. Intuitively, $q_i < 0.5$ indicates that worker j_i is more likely to give an incorrect answer than a correct one. Thus, we can either simply ignore this worker in the jury selection process, or modify her answer according to the specific voting strategy. For example, for MV, we can regard vote 0 as 1 and vote 1 as 0 if the vote is given by a worker whose quality is less than 0.5; for BV, according to its definition, it can reinterpret the vote given by a worker with quality $q_i < 0.5$ as an opposite vote given by a worker with quality $1 - q_i > 0.5$.[1] Moreover, in our experiments with real human workers, we observed that their qualities were generally well above 0.5. We thus assume that $q_i \geq 0.5$ in our subsequent discussions, without loss of generality.

4. COMPUTING JURY QUALITY FOR OPTIMAL STRATEGY

In the previous section, we have proved that BV is the optimal voting strategy with respect to JQ. And thus, in order to solve JSP, we only need to figure out J^* such that $JQ(J^*, BV, \alpha)$ is maximized. An immediate question is whether $JQ(J, BV, \alpha)$ can be computed efficiently. Unfortunately, we find that computing $JQ(J, BV, \alpha)$ is NP-hard (Section 4.1). To alleviate this, we propose an efficient approximation algorithm with theoretical bounds to compute JQ for BV in this section.

4.1 NP-hardness of computing $JQ(J, BV, \alpha)$

Note that [7] has previously proposed an efficient algorithm to compute $JQ(J, MV, 0.5)$ in $\mathcal{O}(n \log n)$. However, this polynomial algorithm cannot be adapted to compute JQ for BV. The main reason is that computing JQ for BV is an NP-hard problem.

THEOREM 2. *Given α and J, computing JQ for BV, or $JQ(J, BV, \alpha)$, is NP-hard.*

The proof is non-trivial and we present the detailed proof in the technical report [15] due to space limits. The idea of the proof is that the partition problem [32] (a well-known NP-complete problem) can be reduced to the problem of computing $JQ(J, BV, 0.5)$ for some J. Hence, the computation of $JQ(J, BV, 0.5)$ is not easier than the partition problem. Moreover, computing $JQ(J, BV, 0.5)$ is not in NP (it is not a decision problem), which makes the problem of computing $JQ(J, BV, \alpha)$ for $\alpha \in [0, 1]$ NP-hard.

To avoid this hardness result, we propose an approximation algorithm. We first discuss the computation of $JQ(J, BV, 0.5)$ in Section 4.2 and 4.3, and give its approximation error bound in Section 4.4. Finally, we briefly discuss how to adapt the algorithm to $\alpha \in [0, 1]$ in Section 4.5.

$R(V) = u(V) - w(V)$	$A_0(V)$	$A_1(\overline{V})$	$A_0(V) + A_1(\overline{V})$
$R(V) > 0$	$e^{u(V)}/2$	$e^{u(V)}/2$	$e^{u(V)}$
$R(V) = 0$	$e^{u(V)}/2$	0	$e^{u(V)}/2$
$R(V) < 0$	0	0	0

Figure 3: Expressing $A_0(V) + A_1(\overline{V})$ using $R(V)$ and $u(V)$

4.2 Analysis of Computing $JQ(J, BV, 0.5)$

Let us first give some basic analysis for computing $JQ(J, BV, 0.5)$ before we introduce our approximation algorithm. To facilitate our analysis, we first define a few symbols.

- $A_0(V) = 0.5 \cdot \Pr(V \mid \mathbf{t} = 0) \cdot \mathbb{1}_{\{BV(V)=0\}}$;
- $A_1(V) = 0.5 \cdot \Pr(V \mid \mathbf{t} = 1) \cdot \mathbb{1}_{\{BV(V)=1\}}$;
- $\overline{V} = \{\bar{v}_1, \bar{v}_2, \ldots, \bar{v}_n\}$, where $\bar{v}_i = 1 - v_i$ $(1 \leq i \leq n)$.

From Figure 2 we observe that $A_0(V) = A_1(\overline{V})$. For example, $A_0(\{0, 1, 0\}) = A_1(\{1, 0, 1\}) = 0.108$ and $A_0(\{1, 0, 1\}) = A_1(\{0, 1, 0\}) = 0$. The observation motivates us to consider $A_0(V)$ and $A_1(\overline{V})$ together, and we can prove that

$$\begin{aligned} JQ(J, BV, 0.5) &= \sum\nolimits_{V \in \Omega} [\, A_0(V) + A_1(V) \,] \\ &= \sum\nolimits_{V \in \Omega} [\, A_0(V) + A_1(\overline{V}) \,], \end{aligned} \tag{5}$$

as $V \to \overline{V}$ defines a one-to-one correspondence between Ω and Ω.

We further define $u(V)$ and $w(V)$ as follows.

$$u(V) = \ln \Pr(V | \mathbf{t} = 0) = \sum\nolimits_{i=1}^{n} [\, (1 - v_i) \ln q_i + v_i \ln(1 - q_i) \,],$$

$$w(V) = \ln \Pr(V | \mathbf{t} = 1) = \sum\nolimits_{i=1}^{n} [\, v_i \ln q_i + (1 - v_i) \ln(1 - q_i) \,],$$

Let $R(V) = u(V) - w(V)$ and $\sigma(q_i) = \ln \frac{q_i}{1-q_i}$ (as $q_i \geq 0.5$, $\sigma(q_i) \geq 0$), we have

$$R(V) = \sum_{i=1}^{n} [\, (1 - 2v_i) \cdot \sigma(q_i) \,], \;\; e^{u(V)} = \prod_{i=1}^{n} q_i^{(1-v_i)} \cdot (1 - q_i)^{v_i}. \tag{6}$$

As illustrated in Figure 3, we can express $A_0(V) + A_1(\overline{V})$ based on the sign of $R(V)$ and the value of $u(V)$. And therefore,

$$JQ(J, BV, 0.5) = \sum_{V \in \Omega} [\mathbb{1}_{\{R(V)>0\}} \cdot e^{u(V)} + \mathbb{1}_{\{R(V)=0\}} \cdot \tfrac{e^{u(V)}}{2}].$$

Motivated by the above formula[2], we can apply an iterative approach which expands J with one more worker at each iteration and thus compute $JQ(J, BV, 0.5)$ in n total iterations. In the k-th iteration, we consider $V^k \in \{0, 1\}^k$. We aim to construct a map structure with $(key, prob)$ pairs, where the domain of key is

[1]Details of the reinterpretation can be found in the technical report [15].

[2]Note that the reason why $A_0(V) \neq A_1(\overline{V})$ when $u(V) = w(V)$ is that as $0.5 \cdot e^{u(V)} = 0.5 \cdot e^{w(V)}$, based on Theorem 1, $BV(V) = BV(\overline{V}) = 0$, so $A_0(V) = 0.5 \cdot e^{u(V)}$ and $A_1(\overline{V}) = 0$.

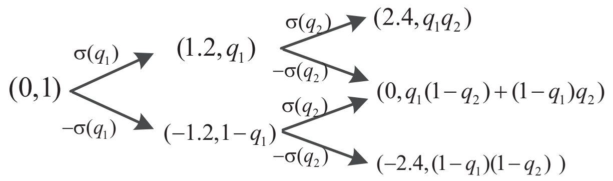


Figure 4: The iterative approach { pair $(key, prob)$ }

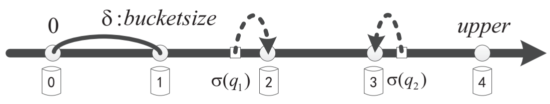


Figure 5: Principle of the bucket array.

$\{\ R(V^k)\ |\ V^k \in \{0,1\}^k\ \}$, and the corresponding value of the key, or $prob$ is

$$prob = \sum_{R(V^k)=key\ \wedge\ V^k\in\{0,1\}^k} e^{u(V^k)}\ . \tag{7}$$

Suppose in the k-th iteration, such a map structure is constructed. Then in the next iteration, we can generate a new map structure from the old map structure: for each $(key, prob)$ in the old map structure, based on the possible choices of v_{k+1} and by considering two formulas in Equation 6, we have

1. for $v_{k+1} = 0$, the new key $key + \sigma(q_{k+1})$ is generated and $prob \cdot q_{k+1}$ is added to the prob of the new key;

2. for $v_{k+1} = 1$, the new key $key - \sigma(q_{k+1})$ is generated and $prob \cdot (1 - q_{k+1})$ is added to the prob of the new key.

EXAMPLE 4. *We give an example to illustrate the above process in Figure 4, where $n = 2$ and $\sigma(q_1) = \sigma(q_2) = 1.2$. Starting from $(0, 1)$, for $v_1 = 0$ and $v_1 = 1$, it respectively creates $(\sigma(q_1) : q_1)$ and $(-\sigma(q_1) : (1 - q_1))$ in the first iteration. Then it leverages the stored $(key, prob)$ pair to generate new pairs in the second iteration by considering different v_2. Note that as $\sigma(q_1) = \sigma(q_2)$, if $(\sigma(q_1), q_1)$ takes $v_2 = 1$ and $(-\sigma(q_1), (1-q_1))$ takes $v_2 = 0$, then they go to the same $key = 0$, and their new prob, $q_1 \cdot (1 - q_2)$ and $(1 - q_1) \cdot q_2$ are added together.*

4.3 Bucket-Based Approximation Algorithm

By our intractability result for JQ we know that the domain of keys, or $\{R(V) \mid V \in \{0,1\}^n\}$ is exponential. In order to address this issue, we set a controllable parameter *numBuckets* and map $\sigma(q_i)$ to a bucket integer $b_i \in [\,0, numBuckets\,]$, where the interval between adjacent buckets, called bucketsize (denoted as δ) is the same. Suppose $numBuckets = d \cdot n$, i.e., a constant multiple of the number of jury members, then, for each iteration, the number of possible values in the key is bounded by $2dn^2 + 1$ (in the range $[-dn^2, dn^2]$) Considering all n iterations, the time complexity is bounded by $\mathcal{O}(dn^3)$, which is of polynomial order.

We detail this process in Algorithm 1. To start with, the function `GetBucketArray` assigns b_i to worker j_i based on $\sigma(q_i)$. The computation of b_i proceeds as follows. At first, we fix a range $[0, upper]$ where $upper = \max_{i\in[1,n]}\ \{\sigma(q_i)\}$. Then, we divide the range into $numBuckets$ of buckets with equal length, denoted by $\delta = \frac{upper}{numBuckets}$. Finally, each worker j_i's bucket number b_i is assigned to its closet bucket: $b_i = \left\lceil \frac{\sigma(q_i)}{\delta} - \frac{1}{2} \right\rceil$. Figure 5 illustrates an example where $numBuckets = 4$. Since $\sigma(q_1)$ is the closet to bucket number 2, so $b_1 = 2$, and similarly $b_2 = 3$.

Algorithm 1 `EstimateJQ`

```
Input:  $J = \{j_1, j_2 \cdots j_n\}$, $numBuckets$ , $n$
Output: $\widehat{JQ}$
 1: $b$ = GetBucketArray($J$, $numBuckets$, $n$);
 2: $b$ = Sort($b$); // sort in decreasing order, for pruning
 3: $J$ = Sort($J$); // sort based on worker quality, similar as above
 4: $aggregate$ = AggregateBucket($b$, $n$); // for pruning
 5: $\widehat{JQ} = 0$; // estimated JQ
 6: $SM[\,0\,] = 1$; //initialize a map structure
 7: for $i = 1$ to $n$ do
 8:     $M = map()$; //initialize an empty map structure
 9:     for $(key, prob) \in SM$ do
10:         $flag, value$ = Prune($key, prob, aggregate[i]$);
11:         if $flag$ = true then
12:             $\widehat{JQ} += value$;
13:             continue // for pruning
14:         if $key + b[i] \notin M$ then
15:             $M[\,key + b[i]\,] = 0$;
16:         $M[\,key + b[i]\,] += prob \cdot q_i$; // for $v_i = 0$
17:         if $key - b[i] \notin M$ then
18:             $M[\,key - b[i]\,] = 0$;
19:         $M[\,key - b[i]\,] += prob \cdot (1 - q_i)$; // for $v_i = 1$
20:     $SM = M$;
21: for $(key, prob) \in SM$ do
22:     if $key > 0$ then
23:         $\widehat{JQ} += prob$;
24:     if $key = 0$ then
25:         $\widehat{JQ} += 0.5 \cdot prob$;
26: return $\widehat{JQ}$;
```

Algorithm 2 `Pruning Techniques`

```
def AggregateBucket($b, n$):
    $aggregate = [\,0, 0 \cdots 0\,]$; // $n$ elements, all 0
    for $i = n$ to 1 do
        if $i = n$ then
            $aggregate[i] = b[i]$;
        else
            $aggregate[i] = aggregate[i+1] + b[i]$;
    return $aggregate$

def Prune($key, prob, number$):
    $flag$ = false;
    if $key > 0$ and $key - number > 0$ then
        $flag$ = true; $value = prob$;
    if $key < 0$ and $key + number < 0$ then
        $flag$ = true; $value = 0$;
    return $flag$, $value$;
```

After mapping each worker to a bucket b_i, we iterate over n workers (step 7-20). For a given worker j_i, based on each $(key, prob)$ pair in the stored map SM, we update key and $prob$, based on two possible values of v_i (steps 14-19)[3] in the new map M. SM will then be updated as the newly derived map M for next iteration (step 20). Finally, the $(key, value)$ pairs in SM are used in the evaluation of the Jury Quality (steps 21-25), based on the cases in Figure 3.

Pruning Techniques. We can further improve the running time of the approximation algorithm by applying some pruning techniques in Algorithm 2, in order to prune redundant computations. For example, assume $n = 5$, and the derived $b = [3, 7, 4, 3, 2]$. In the second iteration, consider the $key = 3 + 7 = 10$ ($v_1 = 0$ and

[3]Note that as we only care about the sign ($+$, 0 or $-$) of $R(V)$, and we approximate $\sigma(q_i)$ as $\delta \cdot b_i$, we can map $\sigma(q_i)$ to b_i and add/subtract the integer b_i.

$v_2 = 0$). No matter what the rest of the three votes are, the aggregated buckets cannot be negative (since $4+3+2 = 9 < 10$), so we can safely prune the search space for $key = 10$ (which takes $2^3 = 8$ computations). To further increase the efficiency, in Algorithm 2 we first sort the bucket array and J in decreasing order (step 2-3), guaranteeing that the highest bucket is considered first, and then compute the *aggregate* array via `AggregateBucket` (step 4), which makes the pruning phase (step 10-13) more efficient. The function `Prune` uses *aggregate* to decide whether to prune or not.

4.4 Approximation Error Bound

Let $\widehat{JQ}$ denote the estimated value returned by Algorithm 1, and JQ denote the real Jury Quality. We evaluate the *additive error* bound on $|JQ - \widehat{JQ}|$ and we can prove that:

$$\widehat{JQ} \le JQ \quad \text{and} \quad JQ - \widehat{JQ} < e^{\frac{n\delta}{4}} - 1, \tag{8}$$

where n is the jury size and $\delta = \frac{upper}{d \cdot n}$ is the bucketsize. Interested readers can refer to technical report [15] for the detailed proof.

We next show that the bound is very small ($< 1\%$ by setting $d \ge 200$) in real cases. First we notice that (i) $\sigma(q)$ is a strictly increasing function and (ii) $\sigma(0.99) < 5$. So let us assume $upper < 5$. We can safely make the assumption, since if not, there exists a worker of quality $q_i > 0.99$, and then $JQ \in (0.99, 1]$, as Lemma 1 will show. Thus we can just return $\widehat{JQ} = q_i > 0.99$, which makes $JQ - \widehat{JQ} < 1\%$. After dividing the interval $[0, upper]$ into $d \cdot n$ equal buckets, we have $\delta < \frac{5}{d \cdot n}$. Using this δ bound in Equation 8, we have $JQ - \widehat{JQ} < e^{\frac{5}{4 \cdot d}} - 1$. By setting $d \ge 200$, the bound is $JQ - \widehat{JQ} < 0.627\% < 1\%$.

4.5 Incorporation of Prior

In the previous section, we have assumed a prior $\alpha = 0.5$. Here, we drop this assumption and show how we can adapt our approaches to a generalized prior $\alpha \in [0, 1]$, given by the task provider. By Theorem 3, it turns out this is equivalent to computing $JQ(J', BV, 0.5)$, where J' is obtained by adding a worker (with quality α) to J:

THEOREM 3. *Given α and J, $JQ(J, BV, \alpha) = JQ(J', BV, 0.5)$, where $J' = J \cup \{j_{n+1}\}$ and $q_{n+1} = \alpha$.*

Due to lack of space, interested readers can refer to technical report [15] for the detailed proof.

Thus we can use Algorithm 1 for any prior α, by adding to the jury a pseudo-worker of quality α. Moreover, the approximation error bound proved in Section 4.4 also holds.

In summary, to compute $JQ(J, BV, \alpha)$, we have developed an approximation algorithm with time complexity $\mathcal{O}(d \cdot n^3)$, with an additive error bound within 1%, for $d \ge 200$.

5. JURY SELECTION PROBLEM (JSP)

Now we focus on addressing $J^* = \arg\max_{J \in \mathcal{C}} JQ(J, BV, \alpha)$, for $\mathcal{C}$, the set of all feasible juries (i.e., $\mathcal{C} = \{J \mid J \subseteq W \wedge \sum_{i=1}^{n} c_i \le B\}$).

Before formally addressing JSP, we turn our attention to two monotonicity properties of $JQ(J, BV, \alpha)$: with respect to varying the jury size ($|J|$), and with respect to a worker j_i's quality (q_i). These properties can help us solve JSP under certain cost constraints.

LEMMA 1 (MONOTONICITY ON JURY SIZE). *Given α and J , $JQ(J, BV, \alpha) \le JQ(J', BV, \alpha)$, where $J' = J \cup \{j_{n+1}\}$.*

LEMMA 2 (MONOTONICITY ON WORKER QUALITY). *Given α and J. Let $J' = J$ except that $q'_{i_0} \ge q_{i_0} \ge 0.5$ for some i_0, then $JQ(J', BV, \alpha) \ge JQ(J, BV, \alpha)$.*

PROOF. Due to space limits, interested reader can refer to technical report [15] about the proofs for Lemma 1 and 2. □

A direct consequence of Lemma 1 is that "the more workers, the better JQ for BV". So for the case that each worker will contribute voluntarily ($c_i = 0$ for $1 \le i \le N$) or the budget constraint satisfies on all subsets of the candidate workers W (i.e., $B \ge \sum_{i=1}^{N} c_i$), we can select all workers in W.

Lemma 2 shows that a worker with higher quality contributes not less in JQ compared with a lower quality worker. For the case that each worker has the same cost requirement c, i.e., $c_i = c_j = c$ for $i, j \in [1, N]$, we can select the top-k workers sorted by their quality in decreasing order, where $k = \min\left\{\left\lfloor \frac{B}{c} \right\rfloor, N\right\}$.

Although the above two properties can indicate us to solve JSP under certain conditions, the case for JSP with arbitrary individual cost is much more complicated as we have to consider not only the worker j_i's quality q_i, but also her cost c_i, and both may vary between different workers.

We can formally prove that JSP is NP-hard in Theorem 4. Note that JSP, in general, is NP-hard due to the fact that it cannot avoid computing $JQ(J, BV, \alpha)$ at each step, which is an NP-hard problem itself. Moreover, even if we assume the existence of a polynomial oracle for computing $JQ(J, BV, \alpha)$ (e.g., Algorithm 1), the problem still remains NP-hard, as we can reduce a n-th order Knapsack Problem [7] to it. Interested readers can refer to the technical report [15] for more details.

THEOREM 4. *Solving JSP is NP-hard.*

5.1 Heuristic Solution

To address the computational hardness issue, we use the *simulated annealing heuristic* [19], which is a stochastic local search method for discrete optimization problems. This method can escape local optima and is proved to be effective in solving a variety of computationally hard problems [5,10].

The simulated annealing heuristic mimics the cooling process of metals, which converge to a final, "frozen" state. A temperature parameter T is used and iteratively reduced until it is small enough. For a specific value of T, the heuristic performs several local neighbourhood searches. There is an objective value on each location, and let Δ denote the difference in objective value between the searched location and the original location. For each local search, the heuristic makes a decision whether to "move" to the new location or not based on T and Δ:

1. if the move will not decrease the objective value (i.e., $\Delta \ge 0$), then the move is accepted;
2. if the move will decrease the objective value (i.e., $\Delta < 0$), the move is accepted with probability $\exp(-\frac{\Delta}{T})$, i.e., by sampling from a Boltzmann distribution [21].

The reason for not immediately rejecting the move towards a worse location is that it tries to avoid getting stuck in local optima. Intuitively, when T is large, it is freer to move than at lower T. Moreover, a large Δ restricts the move as it increases the chances of finding a very bad case.

We can apply the simulated annealing heuristic to solve JSP in Algorithm 3 by assuming that each location is a jury set $J \subseteq W$ and its objective value is $JQ(J, BV, \alpha)$. What is important in simulated annealing is the design of local search. Before introducing

Algorithm 3 JSP

```
Input: $W = \{j_1, j_2, \dots, j_N\}$, $B$, $N$
Output: $\widehat{J}$
 1: $T = 1.0$; // initial temperature parameter
 2: $X = [\, x_1 = 0, x_2 = 0, \dots, x_N = 0 \,]$; // all initialized as 0
 3: $\widehat{J} = \emptyset$; // estimated optimal jury set $J^*$
 4: $M = 0$; // the overall monetary incentive for selected workers
 5: $H = \emptyset$; // the set containing indexes for selected workers
 6: while $T \geq \epsilon$ do
 7:     for $i = 1$ to $N$ do
 8:         randomly pick an index $r \in \{1, 2, \dots, N\}$;
 9:         if $x_r = 0$ and $M + c_r \leq B$ then
10:             $x_r = 1$;  $M = M + c_r$;
11:             $\widehat{J} = \widehat{J} \cup \{j_r\}$;  $H = H \cup \{r\}$;
12:         else
13:             $X, M, \widehat{J}, H = \texttt{Swap}(X, M, \widehat{J}, H, r, B, N)$;
14:     $T = T/2$; // cool the temperature
15: return $\widehat{J}$;
```

Algorithm 4 Swap

```
Input: $X$, $M$, $\widehat{J}$, $H$, $r$, $B$, $N$
Output: $X$, $M$, $\widehat{J}$, $H$
 1: if $x_r = 0$ then
 2:     randomly pick an index $k \in H$;
 3:     $a = k$ ; $b = r$ ; // store the index
 4: else
 5:     randomly pick an index $k \in \{1, 2, \dots, N\} \backslash H$;
 6:     $a = r$ ; $b = k$ ; // store the index
 7: if $M - c_a + c_b \leq B$ then
 8:     $\Delta = \texttt{EstimateJQ}(\, \widehat{J} \setminus \{j_a\} \cup \{j_b\} \,) - \texttt{EstimateJQ}(\widehat{J})$;
 9:     if $\Delta \geq 0$ or $random(0,1) \leq \exp(-\frac{\Delta}{T})$ then
10:         $x_a = 0$;  $x_b = 1$;  $M = M - c_a + c_b$;
11:         $\widehat{J} = \widehat{J} \setminus \{j_a\} \cup \{j_b\}$;  $H = H \backslash \{a\} \cup \{b\}$;
12: return $X$, $M$, $\widehat{J}$, $H$
```

our design of local search, we first explain some variables to keep in Algorithm 3: H is used to store the indexes of selected workers, M is used to store their aggregated cost, and $X = [x_1, x_2, \dots, x_N]$ is used to keep the current state of each worker ($x_i = 1$ indicates that worker j_i is selected and 0 otherwise). Starting from an initial X, we iteratively decrease T (step 14) until T is small enough (step 6). In each iteration, we perform N local searches (steps 7-13), by randomly picking an index r out of the N worker indexes. Based on the randomly picked x_r, we either select the worker if adding the worker does not violate the budget B (steps 9-11), or execute Swap, which is described in Algorithm 4. The decision to swap is made based on different x_r values:

1. if $x_r = 0$, a randomly picked worker $k \in H$ is replaced with worker r if the replacement does not violate the budget constraint and the move is accepted based on Δ and T;

2. if $x_r = 1$, the algorithm performs similarly to the above case, and it replaces worker r with a randomly picked worker $k \in \{1, 2, \cdots, N\} \backslash H$ if the budget constraint still satisfies and the move is accepted as above.

While the heuristic does not have any bound on the returned jury ($\widehat{J}$) versus the optimal jury (J^*), we show in the experiments (Section 6) that it is close to the optimal by way of comparing the real and estimated JQ (i.e., $JQ(\widehat{J}, BV, \alpha)$ and $JQ(J^*, BV, \alpha)$).

6. EXPERIMENTAL EVALUATION

In this section we present the experimental evaluation of JQ and JSP, both on synthetic data and real data. For each dataset, we first evaluate the solution to JSP first, and then give detailed analysis on the computation of JQ. The algorithms were implemented in Python 2.7 and evaluated on a 16GB memory machine with Windows 7 64bit.

6.1 Synthetic Dataset

6.1.1 Setup

First, we describe our default settings for the experiments. Similar to the settings in [7], we generate each worker j_i's quality q_i and cost c_i via Gaussian distributions, i.e., $q_i \sim \mathcal{N}(\mu, \sigma^2)$ and $c_i \sim \mathcal{N}(\widehat{\mu}, \widehat{\sigma}^2)$. We also set parameters following [7], i.e., $\mu = 0.7$, $\sigma^2 = 0.05$, $\widehat{\mu} = 0.05$ and $\widehat{\sigma}^2 = 0.2$. By default, $B = 0.5$, $\alpha = 0.5$ and the number of candidate workers in W is $N = 50$. For JSP (Algorithm 3), we set $\epsilon = 10^{-8}$; for JQ computation (Algorithm 1), we set $numBuckets = 50$. To achieve statistical significance of our results, we repeat the results 1,000 times and report the average values.

6.1.2 System Comparison

We first perform the comparison of JSP with previous works, in an end-to-end system experiment. Cao et al. [7] is the only related algorithm we are aware of, which solves JSP under the MV strategy in an efficient manner. Formally, it addresses JSP as $\arg\max_{J \in \mathcal{C}} JQ(J, MV, 0.5)$. We denote their system as *MVJS* (Majority Voting Jury Selection System) and our system (Figure 1) as *OPTJS* (Optimal Jury Selection System). We compare the two systems by measuring the JQ on the returned jury sets.

The results are presented in Figure 6. We first evaluate the performance of the two systems by varying $\mu \in [0.5, 1]$ in Figure 6(a), which shows that *OPTJS* always outperforms *MVJS*, and *OPTJS* is more robust with low-quality workers. For example, when $\mu = 0.6$, the JQ of *OPTJS* leads that of *MVJS* for 5%. By fixing $\mu = 0.7$, Figure 6(b)-(d) respectively vary $B \in [0.1, 1]$, $N \in [10, 100]$, $\widehat{\sigma} \in [0.1, 1]$ and compare the performance of *MVJS* and *OPTJS*, which all show that *OPTJS* consistently performs better than *MVJS*. In Figure 6(b), *OPTJS* on average leads around 3% compared with *MVJS* for different B; in Figure 6(c), *OPTJS* is better than *MVJS*, especially when the number of workers is limited (say when $n = 10$, *OPTJS* leads *MVJS* for more than 6%); in Figure 6(d), compared with *MVJS*, *OPTJS* is more robust with the change of $\widehat{\sigma}$.

In summary, *OPTJS* always outperforms *MVJS* and, moreover, it is more robust with (1) lower-quality workers, (2) limited number of workers and (3) different cost variances.

6.1.3 Evaluating OPTJS

Next, we test the approximation error of Algorithm 3 by fixing $N = 11$ and varying $B \in [0.05, 0.5]$. Because of its NP-hardness, J^* is obtained by enumerating all feasible juries. We record the optimal $JQ(J^*, BV, 0.5)$ and the returned $JQ(\widehat{J}, BV, 0.5)$ in Figure 7(a). It shows that the two curves almost coincide with each other. As mentioned in Section 6.1.1, each point in the graph is averaged over repeated experiments. Thus, we also give statistics of the difference $JQ(J^*, BV, 0.5) - JQ(\widehat{J}, BV, 0.5)$ on all the 10,000 experiments considering different B (B changes in $[0.05, 0.5]$ with step size 0.05) in Table 3, which shows that more than 90% of them have a difference less than 0.01% and the maximum error is within 3%.

Our next experiment is to test the efficiency of Algorithm 3. We set $B = 0.5$ and vary $N \in [100, 500]$. The results are shown in Figure 7(b). We observe that the running time increases linearly with N, and it is less that 2.5 seconds even for high numbers of workers ($N = 500$). It is fairly acceptable in real situations as the JSP can be done offline.

Table 3: Counts in different error ranges

%	$[0, 0.01]$	$(0.01, 0.1]$	$(0.1, 1]$	$(1, 3]$	$(3, +\infty)$
Counts	9301	231	408	60	0

6.1.4 *JQ Computation*

We now turn our attention to the computation of JQ, which is an essential part of *OPTJS*. We denote here by n the jury size.

We first evaluate the optimality of BV with respect to JQ. Due to the fact the computing JQ in general is NP-hard, we set $n = 11$ and evaluate JQ for four different strategies: two deterministic ones (MV-Majority Voting, and BV-Bayesian Voting), and two randomized ones (RBV-Random Ballot Voting[4] and RMV-Randomized Majority Voting). We vary $\mu \in [0.5, 1]$ and illustrate the resulting JQ in Figure 8(a). It can be seen that the JQ for BV outperforms the others. Moreover, unsurprisingly, all strategies have their worst performance for $\mu = 0.5$ as the workers are purely random in that case. But when $\mu = 0.5$, BV also performs robust (with JQ 93.3%), the reason is that other strategies are sensitive to low-quality workers, while BV can wisely decides the result by leveraging the workers' qualities. Finally, the randomized version of MV, i.e., RMV, performs not better than MV for $\mu \geq 0.5$, as randomized strategies may improve the error bound in the worst case [23]. The JQ under RBV always keeps at 50% since it is purely random.

To further evaluate the performance of different strategies for different jury sizes, and for a fixed $\mu = 0.7$, we vary $n \in [1, 11]$ and plot the resulting qualities in Figure 8(b). The results show that as n increases, the JQ for the two randomized strategies stay the same and BV is the highest among all strategies. To be specific, when $n = 7$, the BV is about 10% better than MV. In summary, BV performs the best among all strategies.

Having compared the JQ between different strategies, we now focus on addressing the computation of JQ for BV, i.e., $JQ(J, BV, 0.5)$ in Figure 9. We first evaluate the effect of the quality variance σ^2 with varying mean μ in Figure 9(a). It can be seen that JQ has the highest value for a high variance when $\mu = 0.5$. It's because under a higher variance, worker qualities are more likely to deviate from the mean (0.5), and so, it's likely to have more high-quality workers.

Then we address the effectiveness of Algorithm 1 for approximating the real JQ. We first evaluate the approximation error in Figure 9(b) by varying $numBuckets \in [10, 200]$. As can be seen, the approximation error drops significantly with $numBuckets$, and is very close to 0 if we have enough buckets. In Figure 9(c) we plot the histogram of differences between the accurate JQ and the approximated JQ (or $JQ - \widehat{JQ}$) over all repeated experiments by setting $numBuckets = 50$. It is heavily skewed towards very low errors. In fact, the maximal error is within 0.01%.

Finally, we evaluate the computational savings of the pruning techniques of Algorithm 1 by varying the number of workers $n \in [100, 500]$ in Figure 9(d). The pruning technique is indeed effective, saving more than half the computational cost. Moreover, it scales very well with the number of workers. For example, when $n = 500$, the estimation of JQ runs within 2.5s without pruning technique, while finishing within 1s facilitated by the proposed pruning methods.

6.2 Real Dataset

6.2.1 *Dataset Collection*

We collected the real world data from the Amazon Mechanical Turk (AMT) platform. AMT provides APIs and allows users to batch multiple questions in Human Intelligence Tasks (HIT). Each worker is rewarded with a certain amount of money upon completing a HIT. The API also allows to set the number of assignments (denoted m) to a HIT, guaranteeing it can be answered m times by different workers. To generate the HITs, we use the public sentiment analysis dataset[5], which contains 5,152 tweets related to various companies. We randomly select 600 tweets from them, and generate a HIT for each tweet, which asks whether the sentiment of a tweet is positive or not (decision making task). The ground truth of this question is provided by the dataset. The true answers for *yes* and *no* is approximately equal, so we set the prior as $\alpha = 0.5$.

To perform experiments on AMT, we randomly batch 20 questions in a HIT and set $m = 20$ for each HIT, where each HIT is rewarded \$0.02. After all HITs are finished, we collect a dataset which contains 600 decision-making tasks, and each task is answered by 20 different workers. We give several statistics on the worker answering information. There are 128 workers in total, and each of them has answered on average $\frac{600 \times 20}{128} = 93.75$ questions. Only two workers have answered all questions and 67 workers have answered only 20 questions. We used these answers to compute every worker's quality, which is defined as the proportion of correctly answered questions by the worker in all her answered questions. The average quality for all workers is 0.71. There are 40 workers whose qualities are greater than 0.8, and about 10% whose quality is less than 0.6.

6.2.2 *JSP*

To evaluate JSP, for each question, we form the candidate workers set W by collecting all 20 workers who answered the question, i.e., having $N = |W| = 20$. We follow the settings in experiments on synthetic data except that worker qualities are computed using the real-world data. We then solve JSP for each question by varying $B \in [0.1, 1.0]$, $N \in [3, 20]$ and $\widehat{\sigma} \in [0, 1]$. We compute the average returned JQ by solving JSP for all 600 questions, which is recorded as a point in Figures 10(a)-(c), respectively. It can be seen that Figure 10(a)-(c) has a similar results pattern as Figure 6(b)-(d), i.e., experimental results on the synthetic datasets. Especially, *OPTJS* always outperforms *MVJS* in real-world scenarios.

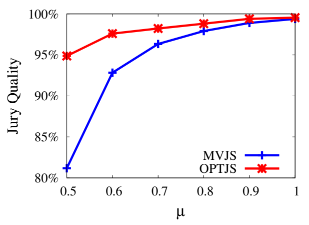
6.2.3 *Is JQ is a good prediction?*

Finally, we try to evaluate whether JQ, defined in Definition 3, is a good way to predict the quality for BV in reality. Notice that, after workers give their votes, we can adopt BV to get the voting result, and then compare it with the true answer of the question. And thus, the goodness of BV in reality can be measured by the "accuracy", which counts the proportion of correctly answered questions according to BV.

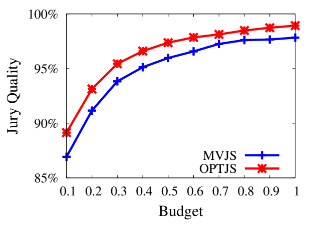
We now test whether JQ is a good prediction of accuracy in reality. For each question, we vary the number of votes (denoted as z). For a given $z \in [0, 20]$, based on the question's answering sequence, we collect its first z votes, then
(i) for each question, knowing the first z workers who answered the question, we can compute the JQ by considering these workers' qualities. Then we take the average of JQ among all 600 questions;
(ii) by considering the first z workers' qualities who answered the question and their votes, BV can decide the result of the question. After that, the accuracy can be computed by comparing voting result and the true answer for each question.

[4]RBV randomly returns 0 or 1 with 50%.

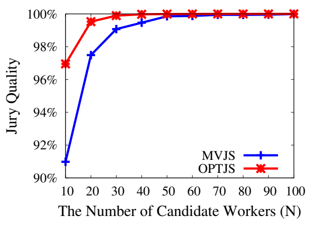
[5]http://www.sananalytics.com/lab/twitter-sentiment/



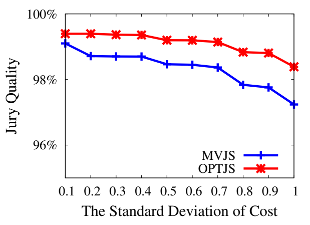
(a) Varying μ



(b) Varying B

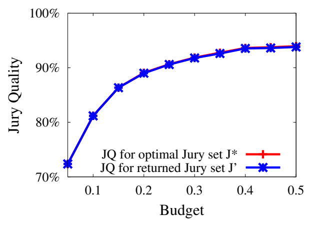


(c) Varying N

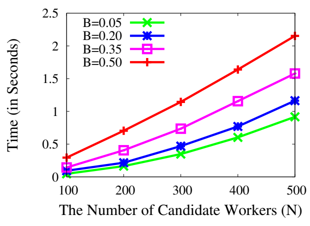


(d) Varying $\widehat{\sigma}$

Figure 6: End-to-end system comparison

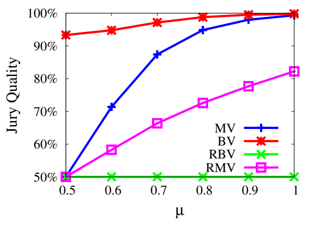


(a) Approximation Error

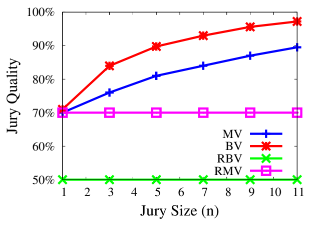


(b) Varying N

Figure 7: Efficiency & Effectiveness of *OPTJS*

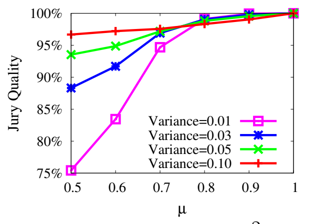


(a) Varying μ

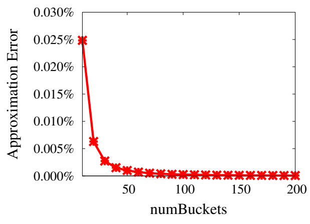


(b) Varying n

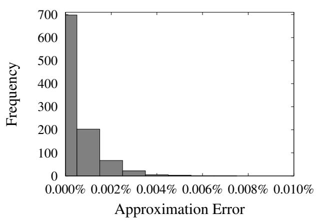
Figure 8: JQ for different strategies



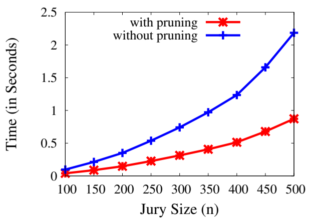
(a) Varying μ and σ^2



(b) Varying $numBuckets$

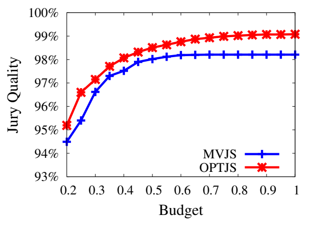


(c) Approximation Error

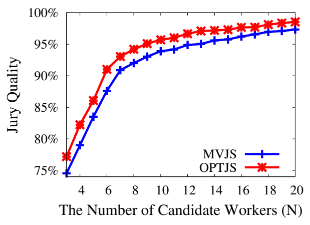


(d) Varying n

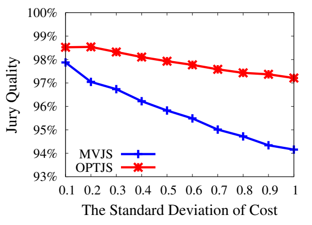
Figure 9: $JQ(J, BV, 0.5)$ computation



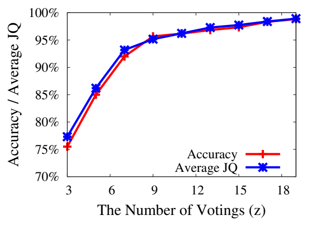
(a) Varying B



(b) Varying N



(c) Varying $\widehat{\sigma}$



(d) Is JQ a good prediction?

Figure 10: Real dataset evaluation

Now given a $z \in [3, 20]$, we compare the average JQ and accuracy in Figure 10(d), which shows that they are highly similar. Hence, it verifies that JQ for BV is really a good prediction of accuracy for BV in reality.

7. EXTENSIONS TO VARIOUS TASK TYPES AND WORKER MODELS

Previously we have talked about how to solve JSP under our data model. Note that we have made two assumptions: (1) it is a decision-making task with binary answer, and (2) each worker's quality is modeled as a constant. However, in reality, it is common for task provider to ask multiple-choice tasks. For example, sentiment analysis tasks [25] require workers to label the sentiment (*positive*, *neutral*, or *negative*) of each task. In addition, the worker's quality can be modeled by measuring the sensitivity and specificity of each possible answer [45], or the *confusion matrix* (CM) [18]. Specifically, a confusion matrix C is a matrix of size $\ell \times \ell$ where each element C_{jk} encodes the probability that the worker votes for k when the true answer is j.

Our proposed algorithms can be easily extended to support these variants. Due to the space limits, we only outline our basic ideas for these extensions, and interested readers are recommended to refer to our technical report [15] for more details.

We first clarify some notations for multiple-choice task. Note that for a task with ℓ possible choices, denoted as $\{0, 1, \ldots, \ell-1\}$, and there exists one unknown true answer [6] $\mathbf{t} \in \{0, 1, \ldots, \ell-1\}$. The domain of the voting from a jury J is $\mathbf{V} \in \Omega = \{0, 1, \ldots, \ell-1\}^n$. Moreover, the prior is now a vector $\vec{\alpha} = \{\alpha_0, \alpha_1, \ldots, \alpha_{\ell-1}\}$ s.t. $\sum_{j=0}^{\ell-1} \alpha_j = 1$.

Following the same solution framework, we first briefly show that BV is still the optimal voting strategy with respect to JQ under this general model, and then sketch how to extend the JQ computation. Finally the extensions for JSP is addressed.

Optimal Strategy Extension:

[6]For the case that each task can have multiple true answers, we can follow [30], which decomposes each task into ℓ decision-making tasks, and publish these ℓ tasks to workers.

To prove the optimality of BV for more general task here, we can follow the same flow as in Section 3.3. Similar to Equation 4, here $\mathbb{E}[\mathbb{1}_{\{S(\mathbf{V})=\mathbf{t}\}}]$ can be expressed as:

$$\sum\nolimits_{V\in\Omega} \sum\nolimits_{t=0}^{\ell-1} \Pr(\,\mathbf{t}=t\,)\cdot \Pr(\,V \mid \mathbf{t}=t\,)\cdot \mathbb{E}[\,\mathbb{1}_{\{S(V)=t\}}\,]. \quad (9)$$

For a given $V \in \Omega$, as

$$(\mathbb{E}[\,\mathbb{1}_{\{S(V)=0\}}\,], \mathbb{E}[\,\mathbb{1}_{\{S(V)=1\}}\,], \ldots, \mathbb{E}[\,\mathbb{1}_{\{S(V)=\ell-1\}}\,])$$

defines a discrete probability distribution, it is not hard to prove that the optimal strategy, or $S^*(V)$ is

$$S^*(V) = \underset{t'\in\{0,1,\ldots,\ell-1\}}{\arg\max}\ \alpha_{t'}\cdot \Pr(V \mid \mathbf{t}=t'), \qquad (10)$$

which corresponds to the Bayes' Theorem [3] that chooses the result as the label t^* with highest posterior probability, i.e., $t^* = \arg\max_{\,t'\in\{0,1,\ldots,\ell-1\}} \Pr(\mathbf{t}=t' \mid V)$. Thus $S^* = BV$.

Jury Quality Computation Extension:

Recall the definition of JQ in Equation 9, to facilitate our understanding, we express $\mathbb{E}[\mathbb{1}_{\{S(\mathbf{V})=\mathbf{t}\}}]$ in the following way

$$\sum\nolimits_{t'=0}^{\ell-1} \alpha_{t'}\cdot [\,\sum\nolimits_{V\in\Omega} \Pr(\,V \mid \mathbf{t}=t'\,)\cdot \mathbb{E}[\,\mathbb{1}_{\{BV(V)=t'\}}\,]\,] \quad (11)$$

This representation enables us to consider each possible true answer separately. For each $t' \in \{0,1,\ldots,\ell-1\}$, we compute

$$H(t') = \sum\nolimits_{V\in\Omega} \Pr(\,V \mid \mathbf{t}=t'\,)\cdot \mathbb{E}[\,\mathbb{1}_{\{BV(V)=t'\}}\,]$$

and then linearly combines the computed $H(t')$ with $\vec{\alpha}$ to get JQ. So the question falls to the computation of $H(t')$.

To compute $H(t')$, for a $V \in \Omega$, we have to keep track of
(1) whether $BV(V) = t'$ or not, and
(2) if $BV(V) = t'$, the value $\Pr(V \mid \mathbf{t} = t')$ should be added.
Similar to the analysis in Section 4.2, we apply an iterative approach, where in each iteration, we expand J with one more worker. We develop a map structure with $(key, prob)$ pairs to store the above two mentioned information. The key is an ℓ-tuple

$$\left(\ln\frac{\Pr(V \mid \mathbf{t}=t')\cdot\alpha_{t'}}{\Pr(V \mid \mathbf{t}=0)\cdot\alpha_0},\ \ldots\ ,\ \ln\frac{\Pr(V \mid \mathbf{t}=t')\cdot\alpha_{t'}}{\Pr(V \mid \mathbf{t}=\ell-1)\cdot\alpha_{\ell-1}} \right)$$

where the i-th element of the tuple is $\ln\frac{\Pr(V\ \mid\ \mathbf{t}=t')\cdot\alpha_{t'}}{\Pr(V\ \mid\ \mathbf{t}=i-1)\cdot\alpha_{i-1}}$. Intuitively, given a $V \in \Omega$, if $BV(V) = t'$, then $\Pr(V \mid \mathbf{t}=t')\cdot\alpha_{t'} \geq \Pr(V \mid \mathbf{t}=t)\cdot\alpha_t$ for any $t \in \{0,1,\ldots,\ell-1\}$, which means that the elements in the stored tuple are all ≥ 0. The value $prob$ corresponding to a key is the aggregated probability $\Pr(V \mid \mathbf{t}=t')$ for the same state. In the k-th iteration, the $V^k = \{0,1,\ldots,\ell-1\}^k$, for a key, we will generate ℓ new keys corresponding to different votes, and update their individual $prob$ for the next iteration. After n iterations, based on identifying the keys whose elements are all ≥ 0, we can get JQ by aggregating the corresponding $prob$s.

Since the values of elements in a tuple are unbounded, we can follow the similar idea in Section 4.3, that is to map each worker's vote to a bucket number. Note that each element in the tuple can be decomposed as the summation of individual worker's vote, thus the number of states in keys are bounded.

Jury Selection Problem Extension:

To address JSP, similarly we can prove that the monotonicity on jury size by extending Lemma 1, which means that "the more workers, the better JQ" still holds for more general case. As the worker is modeled as a confusion matrix (with size $\ell \times \ell$), the extension for Lemma 2 is non-trivial, and it stills remains an open question on what kind of confusion matrix will contribute more to the JQ. Previous works [18,34] have addressed how to rank workers (or to detect spammers in all workers) based on their associated confusion matrices, which may provide good heuristics for us.

For more general cost models where each worker requires arbitrary costs, the simulated annealing heuristic regards computing JQ as a black box, so it can be simply extended here.

8. RELATED WORKS

Crowdsourcing. Nowadays, crowdsourcing has evolved as a problem solving paradigm [6] to address computer-hard tasks. To incorporate the crowd into query processing, crowdsourced databases (e.g., CrowdDB [14], Deco [31], Qurk [27] and CDAS [25]) are built, compared with traditional database systems, they do not hold the closed-world assumption. As a novel paradigm, the power of crowdsourcing has also been leveraged in other applications. For example, in Optical Character Recognition [38], Entity Resolution [39,41], Tagging [43], Schema Matching [17,44], Web Table Understanding [12], Data Cleaning [40] and so on.

Voting Strategy. In order to aggregate the collective wisdom of a jury, given some specific voting of a task from the jury, voting strategies are widely used to return a result, which is an estimation of the ground truth for the task. For example, Majority Voting strategy [7] strictly returns the answer corresponding to higher votes, and Random Ballot Voting [33] randomly selects the returned result. Similarly other strategies [2,9,23–25,28,29] are also talked about in a great deal. Different from their works, here we give a systematic way to classify all the strategies into two categories, and try to observe the optimal strategy in all these strategies under the Jury Selection Problem. Note that different from our problem, people may evaluate strategies under different purposes. For example, [26] analyzes the optimal Bayesian manipulation strategies by assessing the expected loss in social welfare, and [11] applies Bayesian model to take a game-theoretic approach in characterizing the symmetric equilibrium of the game with juries.

Worker Model. To model a worker's quality in crowdsourcing, most existing works [7,25,28,44] define it as a constant parameter indicating the probability that the worker correctly answers a question, while other work [18] defines it as a confusion matrix, which tries to capture relations between labels in questions and is specific to choices in tasks. For the methods to derive worker's quality, a normal way is to leverage the answering history. If they are not sufficient, [25] hides golden questions (questions with known ground truth) and derive the quality based on the worker's answers for them, while other work [18] applies Expectation Maximization [8] algorithm to iteratively updates worker's quality until convergence. For micro-blog services especially in Twitter, the retweet actions are usually explored to derive the error rate for each worker [7]. In our work we define worker's quality by a constant parameter (commonly used by existing works) and assume that they are known in advance. Moreover, we also extend our method to address the confusion matrix mentioned in [18].

Online Processing. There are also some online processing systems [4,16,25] in crowdsourcing, which addresses how to assign tasks to workers and process the workers' answers. For example, [25] proposes quality-sensitive answering model and terminate assigning questions which has got confident answers; [4] proposes an entropy-like approach to define the uncertainty of each question and assigns questions with highest uncertainty; [16] proposes cost-sensitive model to address which questions are better answered by humans or machines. Different from them, we especially evaluate how to estimate the JQ before the workers are selected to answer the questions, and the quality estimation can provide statistics and guidance for the task publisher to wisely invest budget. Even though existing work [25,28] tried to estimate the quality, they assume that each worker is of the same quality.

Expert Team Formation. In social network, several works [13,22] studied the problem of expert team formation, that is, given the aggregated skill requirements for a task, how to find a team of experts with minimum cost (communication cost or individual financial requirement), such that the skill requirements are satisfied. Rather than the skill requirements in [13,22], we focus on the probability of drawing a correct answer, which requires to enumerate exponential number of possibilities and is indeed challenging. In fact we address the Jury Selection Problem, which is firstly proposed by [7]. But we find that the solution is sub-optimal in [7], which cannot leverage the known quality for workers. We formally address the optimal JSP problem in the paper. Some other works [9,35] also talk about how to wisely select sources for integration. The difference is that we assume the workers are given a multiple-label task and the worker model is known, while in their problem setting, the possible answers from different sources are not restricted, and the sources' exact real qualities are unknown in advance.

9. CONCLUSIONS

In this paper, we have studied Jury Selection Problem (JSP) for decision-making tasks, whose objective is to choose a subset of workers, such that the probability of having a correct answer (or Jury Quality, JQ) is maximized. We approach this problem from an optimality perspective. As JQ is related to voting strategy, we prove that an existing strategy, called Bayesian Voting Strategy (BV) is optimal under the JQ. Although computing JQ under BV is NP-hard, we give an efficient algorithm with theoretical guarantees. Moreover, we incorporate the task provider prior information, and we show how to extend JQ computation for different worker models and task types. Finally we evaluate JSP under BV, we prove several properties which can be used for efficient JSP computations under some constraints, and provide an approximate solution to JSP by simulated annealing heuristics.

Acknowledgment

Reynold Cheng, Silviu Maniu, Luyi Mo, and Yudian Zheng were supported by RGC (Project HKU 711110) and HKU (Project 201311159095). We thank the reviewers for their comments.

10. REFERENCES

[1] A.P.Dawid and A.M.Skene. Maximum likelihood estimation of observer error-rates using em algorithm. *Appl.Statist.*, 28(1):20–28, 1979.

[2] Ashish Goel and David Lee, Triadic Consensus: A Randomized Algorithm for Voting in a Crowd. http://arxiv.org/pdf/1210.0664v1.pdf.

[3] Bayes' Theorem. http://en.wikipedia.org/wiki/Bayes'_theorem.

[4] R. Boim, O. Greenshpan, T. Milo, S. Novgorodov, N. Polyzotis, and W. C. Tan. Asking the right questions in crowd data sourcing. In *ICDE*, 2012.

[5] D. Bookstaber. Simulated annealing for traveling salesman problem.

[6] D. Braham. Crowdsourcing as a model for problem solving: An introduction and cases. *Convergence*, 14(1):75–90, 2008.

[7] C. C. Cao, J. She, Y. Tong, and L. Chen. Whom to ask? jury selection for decision making tasks on micro-blog services. *PVLDB*, 5(11):1495–1506, 2012.

[8] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *J.R.Statist.Soc.B*, 30(1):1–38, 1977.

[9] X. L. Dong, B. Saha, and D. Srivastava. Less is more: Selecting sources wisely for integration. *PVLDB*, 6(2):37–48, 2012.

[10] A. Drexl. A simulated annealing approach to the multiconstraint zero-one knapsack problem. *Computing*, 40:1–8, 1988.

[11] J. Duggan and C. Martinelli. A bayesian model of voting in juries. *Games and Economic Behavior*, 37(2):259–294, 2001.

[12] J. Fan, M. Lu, B. C. Ooi, W.-C. Tan, and M. Zhang. A hybrid machine-crowdsourcing system for matching web tables. In *ICDE*, 2014.

[13] F. Farhadi, E. Hoseini, S. Hashemi, and A. Hamzeh. Teamfinder: A co-clustering based framework for finding an effective team of experts in social networks. In *ICDM Workshops*, pages 107–114, 2012.

[14] A. Feng, M. J. Franklin, D. Kossmann, T. Kraska, S. Madden, S. Ramesh, A. Wang, and R. Xin. Crowddb: Query processing with the vldb crowd. *PVLDB*, 4(12):1387–1390, 2011.

[15] Full version. http://i.cs.hku.hk/∼ydzheng2/Jury/Jury.pdf.

[16] J. Gao, X. Liu, B. C. Ooi, H. Wang, and G. Chen. An online cost sensitive decision-making method in crowdsourcing systems. In *SIGMOD*, 2013.

[17] N. Q. V. Hung, N. T. Tam, Z. Miklós, and K. Aberer. On leveraging crowdsourcing techniques for schema matching networks. In *Database Systems for Advanced Applications*, pages 139–154. Springer, 2013.

[18] P. Ipeirotis, F. Provost, and J. Wang. Quality management on amazon mechanical turk. In *SIGKDD workshop*, pages 64–67, 2010.

[19] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, New Series*, 220(4598):671–680, 1983.

[20] A. Lacasse, F. Laviolette, M. Marchand, and F. Turgeon-Boutin. Learning with randomized majority votes. pages 162–177, 2010.

[21] L. Landau and E. Lifshitz. *Statistical Physics. Course of Theoretical Physics 5 (3 ed.)*. Oxford: Pergamon Press, 1980.

[22] T. Lappas, K. Liu, and E. Terzi. Finding a team of experts in social networks. In *KDD*, pages 467–476, 2009.

[23] N. Littlestone and M. K. Warmuth. The weighted majority algorithm. *Inf. Comput.*, 108(2):212–261, Feb. 1994.

[24] X. Liu, X. L. Dong, B. C. Ooi, and D. Srivastava. Online data fusion. *PVLDB*, 4(11):932–943, 2011.

[25] X. Liu, M. Lu, B. C. Ooi, Y. Shen, S. Wu, and M. Zhang. Cdas: A crowdsourcing data analytics system. *PVLDB*, 5(10):1040–1051, 2012.

[26] T. Lu, P. Tang, A. D. Procaccia, and C. Boutilier. Bayesian vote manipulation: Optimal strategies and impact on welfare. In *UAI*, pages 543–553, 2012.

[27] A. Marcus, E. Wu, S. Madden, and R. C. Miller. Crowdsourced databases: Query processing with people. In *CIDR*, pages 211–214, 2011.

[28] L. Mo, R. Cheng, B. Kao, X. S. Yang, C. Ren, S. Lei, D. W. Cheung, and E. Lo. Optimizing plurality for human intelligence tasks. In *CIKM*, 2013.

[29] S. Nitzan and J. Paroush. Collective decision making and jury theorems.

[30] A. G. Parameswaran, H. Garcia-Molina, H. Park, N. Polyzotis, A. Ramesh, and J. Widom. Crowdscreen: algorithms for filtering data with humans. In *SIGMOD*, pages 361–372, 2012.

[31] H. Park, R. Pang, A. G. Parameswaran, H. Garcia-Molina, N. Polyzotis, and J. Widom. Deco: A system for declarative crowdsourcing. *PVLDB*, 2012.

[32] Partition Problem. http://en.wikipedia.org/wiki/Partition_problem.

[33] Random Ballot Voting. http://en.wikipedia.org/wiki/Random_ballot.

[34] V. C. Raykar and S. Yu. Eliminating spammers and ranking annotators for crowdsourced labeling tasks. *Journal of Machine Learning Research*, 2012.

[35] T. Rekatsinas, X. L. Dong, and D. Srivastava. Characterizing and selecting fresh data sources. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 919–930. ACM, 2014.

[36] M. Venanzi, J. Guiver, G. Kazai, P. Kohli, and M. Shokouhi. Community-based bayesian aggregation models for crowdsourcing. In *Proceedings of the 23rd international conference on World wide web*, pages 155–164. International World Wide Web Conferences Steering Committee, 2014.

[37] P. Venetis and H. Garcia-Molina. Quality control for comparison microtasks. In *CrowdKDD, KDD 2012 Workshop*, 2012.

[38] L. von Ahn, B. Maurer, C. McMilen, D. Abraham, and M. Blum. Recaptcha: Human-based character recognition via web security measures. *Science*, 321(5895):1465–1468, 2008.

[39] J. Wang, T. Kraska, M. J. Franklin, and J. Feng. Crowder: Crowdsourcing entity resolution. *PVLDB*, 5(11):1483–1494, 2012.

[40] J. Wang, S. Krishnan, M. J. Franklin, K. Goldberg, T. Kraska, and T. Milo. A sample-and-clean framework for fast and accurate query processing on dirty data. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 469–480, 2014.

[41] J. Wang, G. Li, T. Kraska, M. J. Franklin, and J. Feng. Leveraging transitive relations for crowdsourced joins. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 229–240, 2013.

[42] J. Whitehill, P. Ruvolo, T. Wu, J. Bergsma, and J. R. Movellan. Whose vote should count more: Optimal integration of labels from labelers of unknown expertise. In *NIPS*, pages 2035–2043, 2009.

[43] X. S. Yang, R. Cheng, L. Mo, B. Kao, and D. W. Cheung. On incentive-based tagging. In *ICDE*, pages 685–696, 2013.

[44] C. J. Zhang, L. Chen, H. V. Jagadish, and C. C. Cao. Reducing uncertainty of schema matching via crowdsourcing. *PVLDB*, 6(9):757–768, 2013.

[45] B. Zhao, B. I. Rubinstein, J. Gemmell, and J. Han. A bayesian approach to discovering truth from conflicting sources for data integration. *Proceedings of the VLDB Endowment*, 5(6):550–561, 2012.

Finding the Most Diverse Products using Preference Queries

Orestis Gkorgkas[1], Akrivi Vlachou[2], Christos Doulkeridis[3] and Kjetil Nørvåg[1]
[1]Norwegian University of Science and Technology (NTNU), Trondheim, Norway
[2]Institute for the Management of Information Systems, R.C. "Athena", Athens, Greece
[3]University of Piraeus, Piraeus, Greece
{orestis,vlachou,noervaag}@idi.ntnu.no, cdoulk@unipi.gr

ABSTRACT

In this paper, given a product database and a set of customer preferences, we address the problem of discovering a bounded set of r diverse products that attract the interests of different customers. This problem finds numerous applications in electronic marketplaces, e.g., for selecting the products that are placed in the home page of an online shop. Existing approaches to tackle this problem fall short because they ignore customer preferences, and instead rely solely on products' attributes. We model this problem as a diversity problem, where each product is represented by its reverse top-k result set, and seek r products that maximize their diversity value. Since the problem is NP-hard, we employ a greedy algorithm that takes as input the reverse top-k result sets of all candidate products. To further improve performance, we also design a more efficient approximate algorithm that does not require the computation of all reverse top-k sets. Our experimental evaluation demonstrates the performance of the proposed algorithms and quality of the selected diverse products.

1. INTRODUCTION

Top-k queries [17] help customers select a ranked set of k products that best match their preferences out of an overwhelmingly large collection of products. For a specific customer, her preferences are expressed by means of a top-k query, and highly ranked products in the top-k result are more attractive to the customer. Thus, from the perspective of product sellers, the visibility and the potential market of a product relates to the top-k queries for which the product is highly ranked. Towards this direction, reverse top-k queries [20] retrieve the set of user preferences for which a product appears in their top-k lists. Reverse top-k queries are very important for estimating the impact of the product on the market, as the cardinality of the result set defines an *influence score* [22] for the product, i.e., the number of customers that value a particular product.

In this paper, we study the problem of finding the r most diverse products based on the user preferences. The goal is to find a set of products that are attractive to a wide range of customers with different preferences. For instance, consider an electronic marketplace that wishes to advertise r products on its front page aiming to attract as many new customers as possible. Advertising diverse products that are attractive to different existing customers increases the probability that a new customer finds one of those products attractive. The strategy of advertising the r most influential products [22], i.e., the r products that attract the highest total number of customers, does not necessarily lead to a set of diverse products and may fail to attract many new customers, since such products may be attractive to customers with similar preferences.

User Preferences:

User	$w[1]$	$w[2]$	$w[3]$	**Top-k**
Bob	0.1	0.2	0.7	p_1
Tom	0.1	0.3	0.6	p_1
Jack	0.3	0.1	0.6	p_2
Max	0.8	0.1	0.1	p_3

Products:

Product	$p[1]$	$p[2]$	$p[3]$	**Reverse top-k**
p_1	1	2	6	Bob,Tom
p_2	2	1	6	Jack
p_3	6	5	2	Max

Table 1: Example of product database and user preferences.

Consider for example the set of user preferences and products depicted in Table 1, where maximum values in product attributes are preferable. Assume that the goal is to advertise two products for attracting new customers. Our proposed method selects the $r = 2$ most diverse products based on user preferences, which in our example is the set $\{p_1, p_3\}$. This set is more probable to attract more new customers because p_1 and p_3 satisfy more diverse preferences. For example, a customer with similar preferences to Jack is highly probable to be attracted also to p_1, even though it is not the best option for her on the market. This is because both p_1 and p_2 satisfy users that have high preference for the third dimension (expressed with a high weight $w[3]$). On the other hand, p_3 satisfies users that have totally diverse preferences compared to p_1 and p_2, namely users such as Max that prefer the first dimension.

In this paper, we introduce the problem of finding the r *most diverse* products based on user preferences. The user preferences are captured by the reverse top-k set of each product. We model this problem as a *dispersion* problem [15] using as distance function the dissimilarity of the reverse top-k sets. In this sense, the set of r objects with the maximum diversity is returned to the user. Consequently, the selected objects are appealing to many different customers with dissimilar user preferences. Different from our work, existing solutions for identifying diverse objects rely solely on product attributes and largely overlook user preferences [18]. On the other hand, approaches that identify r objects with high total number of customers [12, 22], often fail to discover truly diverse products that can be appealing to new customers with different preferences than those of the existing ones.

To summarize the contributions of this paper are:

- We study the novel problem of finding the r most diverse products based on user preferences. We model this problem as a *dispersion* problem and define an appropriate distance function that captures the dissimilarity of products based on their reverse top-k sets.
- As dispersion problems are known to be NP-hard [8], we use a greedy algorithm that retrieves r diverse products, after computing the reverse top-k sets of the products efficiently.
- To improve the performance of our algorithm, we propose an alternative algorithm that progressively computes an approximation of the reverse top-k sets of a limited set of candidate products and retrieves a set of r products of high diversity.
- We present maintenance techniques for updating the r most diverse products in the case of dynamic data in a cost-efficient way. In addition, we generalize our approach to support any set-based similarity function.
- We demonstrate the efficiency and achieved diversity of our algorithms using both synthetic and real-life data sets.

The rest of this paper is organized as follows: Section 2 reviews the related work. Section 3 provides the necessary preliminaries, while in Section 4 we formally define the r-Diversity problem. Thereafter, in Section 5, we present a greedy algorithm applied on the reverse top-k sets. In Section 6, we provide a more efficient algorithm that iteratively computes an approximation of the reverse top-k sets and refines the set of most diverse products. Section 7 addresses the case of dynamic data, while Section 8 generalizes our approach for set-based similarity functions. The experimental evaluation is presented in Section 9 and we conclude in Section 10.

2. RELATED WORK

Reverse top-k queries. Vlachou *et al.* first introduced the reverse top-k query in [20, 21]. Two versions of the reverse top-k query were presented, namely monochromatic and bichromatic. Based on the geometrical properties of the monochromatic reverse top-k query, an algorithm for the two dimensional case was proposed. For computing bichromatic reverse top-k queries, an algorithm (called *RTA*) was proposed that exploits the fact that similar queries share common results, in order to avoid evaluating the top-k queries for all user preferences. Thereafter, several papers have studied the problem of efficient reverse top-k computation. An efficient algorithm for the two-dimensional monochromatic reverse top-k that relies on a novel index was proposed in [4]. In [9], efficient evaluation of multiple top-k queries is studied, which in turn enables the computation of the reverse top-k set of a query point. The proposed method avoids evaluating the top-k queries one-by-one by grouping similar queries and evaluate them in a batch. This approach is suitable for processing many reverse top-k queries at once. An approach for processing a large number of continuous top-k queries has appeared in [27]. The proposed framework can be employed to process reverse top-k queries efficiently, however it requires to build an index over the k-th ranked objects of each query that results in high pre-processing cost. Recently, a novel branch-and-bound algorithm for reverse top-k queries has appeared in [23], where both the object data sets and the preferences set are indexed using an R-tree.

Product impact and visibility. Several papers have proposed methods that aim to quantify the impact of products in the market. DADA [11] aims to help manufactures position their products in the market, based on three types of dominance relationship analysis queries. Creating competitive products has been studied in [24]. Customer identification and product positioning has been recently studied in [2], where the attractiveness of a product is defined based on the concept of reverse skyline query. Nevertheless, in these approaches user preferences are expressed as data points that represent preferable products, whereas reverse top-k queries examine user preferences in terms of weighting vectors. Miah *et al.* [14] study a different problem, namely how to select the subset of attributes that increases the visibility of a new product. Product promotion is studied in [25, 26], where the aim is to find the most interesting regions for promotion of a product. Only a few papers have proposed methods for retrieving interesting products by using the reverse top-k queries. In [22], the influence of a product is defined as the size of its reverse top-k set. Then, an algorithm was presented to efficiently retrieve the m most influential products. Discovering k products with maximum number of customers has been studied in [12], where the number of customers is estimated as the size of the reverse top-k set. The problems studied in [12, 22] differ from the diversity problem studied in this paper. Both approaches focus on maximizing the number of existing customers and ignore the similarity of the retrieved reverse top-k sets. These approaches fail to take into account the fact that attracting new customers requires promoting products that are attractive to customers with diverse preferences. Koh *et al.* [10] consider as products packages consisting of multiple components. They study the problem of creating and selecting packages from an existing pool of components such that the number of potential customers is maximized. Similarly to the aforementioned approaches the number of potential customers is estimated using reverse top-k sets, yet they do not study the diversity of the result set.

Diversity in databases. Many approaches have been proposed for retrieving a set of diverse objects. Angel *et al.* [1] study the problem of retrieving k documents relevant to a query q, but are also diverse with each other. The diversity is computed based on document similarity metrics. Drosou *et al.* [7] study the problem of finding the k most

Symbol	Description
$\mathbb{R}^m$	m-dimensional dataspace
S	Set of data objects
D	Subset of S ($D \subseteq S$)
p,q	Data objects/products ($p, q \in S$)
W	Set of weighting vectors
$\mathbf{w}$	A weighting vector ($\mathbf{w} \in W$)
$f_{\mathbf{w}}$	Preference function associated with $\mathbf{w}$
k	Value of top-k
$TOP_k(\mathbf{w})$	Top-k data objects based on $\mathbf{w}$
$RTOP_k(p)$	Reverse top-k result set for object p
$\mathbf{c}_p$	Centroid of vectors in set $RTOP_k(p)$
$d(p, q)$	Cosine distance between centroids $\mathbf{c}_p, \mathbf{c}_q$
$d(\mathbf{u}, \mathbf{v})$	Cosine distance between vectors $\mathbf{u}, \mathbf{v}$
$div(D)$	Diversity value of a set of objects D
D^*	Optimal solution of the r-Diversity problem
$D_r(S)$	Approximate solution of the r-Diversity problem

Table 2: Overview of symbols.

diverse objects in a continuous data stream. DivDB, a system that provides query result diversification, was presented in [19]. Result diversification based on dissimilarity is studied also in [6]. Estimating the diversity of a set of points that fulfill a special property has been studied mainly for selecting representative skyline points. For instance the diversity of two skyline points can be defined as the distance between them [16] or by using their sets of dominated points [13, 18]. More specifically, in [16] the authors define the set of representative skyline to be a set of k objects that maximize the minimum Euclidean distance between any two of the k points. In [13], the representative skyline points are defined based on the distinct number of dominated points. Valkanas *et al.* [18] estimate the diversity of two skyline points by calculating the Jaccard distance of their respective sets of dominated points. The main difference to our work is that the definitions of diversity in the above approaches rely on the attribute values only and cannot exploit the existing user preferences.

3. PRELIMINARIES

Given a space $\mathbb{R}^m$, we assume that we have a set of data objects S where each object $p \in S$ can be represented as an m-dimensional point $p = \{p[1], \ldots, p[m]\}$ where $p[i] \in \mathbb{R}^+$. Each point p can be regarded as an object of a database and each dimension of the point as a specific numerical attribute. Without loss of generality, we assume that larger values are preferable.

Given a scoring function $f : S \to \mathbb{R}^+$,a *top-k* query returns the k objects of S with the best score. The scoring functions used more often, are linear functions of the form $f(p) = \sum_{i=1}^{m} w[i]p[i]$ where $w[i] \geq 0$. Such functions can be represented by an m-dimensional *weighting vector* $\mathbf{w} = \{w[1], \ldots, w[m]\}$. In such cases we denote the function that results from $\mathbf{w}$ as $f_{\mathbf{w}}$. When $\mathbf{w}$ represents the preferences of a user over the objects in S we call this vector *preference vector* or simply *preference.*

Definition 1. ***(Top-k query [20])*** *Given a data set $S \subseteq \mathbb{R}^m$ and a vector $\mathbf{w} \in \mathbb{R}^m$ the result set $TOP_k(\mathbf{w})$ of the top-k query is a set of points such that $TOP_k(\mathbf{w}) \subseteq S$, $|TOP_k(\mathbf{w})| = k$ and $\forall p_1, p_2 : p_1 \in TOP_k(\mathbf{w}), p_2 \in S - TOP_k(\mathbf{w})$ it holds that $f_{\mathbf{w}}(p_1) \geq f_{\mathbf{w}}(p_2)$*

If we have a set of preferences $W \subseteq \mathbb{R}^m$ over a set of products $S \subseteq \mathbb{R}^m$ then for a given product q, we say that the result set of a *reverse top-k* query is a set $RTOP_k(q)$ which consists of all the preference vectors $\mathbf{w}$ for which it holds that $q \in TOP_k(\mathbf{w})$.

Definition 2. ***(Reverse top-k query [20])*** *Given a set of points S, a point p, and a set of vectors W we say that a vector $\mathbf{w}$ belongs in the reverse top-k set $RTOP_k(p)$ of point p, if and only if $\exists q \in TOP_k(\mathbf{w})$ such that $f_{\mathbf{w}}(p) \geq f_{\mathbf{w}}(q)$.*

We can also define the *influence score* of a data object p as the cardinality of the set $RTOP_k(p)$. Table 2 provides an overview of the most basic symbols used in this paper.

4. PROBLEM STATEMENT

Let p and q denote two products (data objects) from a product database S. Also, given a set W of customer preferences (weighting vectors) and an integer k, let $RTOP_k(p) \subseteq W$ and $RTOP_k(q) \subseteq W$ denote the reverse top-k sets of p and q respectively. We also define a distance function $d : S \times S \to \mathbb{R}^+$ as:

$$d(p, q) = f_d(RTOP_k(p), RTOP_k(q))$$

that determines the dissimilarity of any two objects p and q based on their corresponding reverse top-k sets. Notice that this is a radically different approach from existing initiatives that define the distance of two objects based on the objects' attributes only.

The problem of selecting the r most diverse products from a given set S can be viewed as a *dispersion* problem [7, 8, 15, 18], where the aim is to find r objects such that an objective function of their distance d is optimized. The *dispersion sum problem* maximizes the sum of pairwise distances between the r selected products and it has been proved that it is NP-hard by reduction from the clique problem [8].

Problem 1. ***(r-Diversity Problem)***. *Given a set of data objects S and a distance function d measuring the dissimilarity between two objects, the r-Diversity problem is to identify a subset $D^* \subseteq S$ such that:*

$$D^* = \underset{\substack{D \subseteq S \\ |D| = r}}{\arg\max} \sum_{\substack{p,q \in D \\ p \neq q}} d(p, q)$$

The remaining challenge is to define an appropriate function f_d that captures the dissimilarity of the reverse top-k result sets. Hence, the function f_d takes as input two sets of weighting vectors and computes their dissimilarity. We employ a function that relies on the concept of a *centroid* of a set of vectors.

Definition 3. ***(centroid of RTOPk)***. *Given a set of data objects S, a set of weighting vectors W, and an object $p \in S$ such that $RTOP_k(p) \neq \emptyset$, we define as the* centroid *of p the vector:*

$$\mathbf{c}_p = \frac{1}{|RTOP_k(p)|} \sum_{\mathbf{w} \in RTOP_k(p)} \mathbf{w}$$

Since each $RTOP_k$ set corresponds to exactly one data point, the respective centroid corresponds to exactly one data point as well. Therefore each data point can be mapped to exactly one centroid vector and vice-versa.

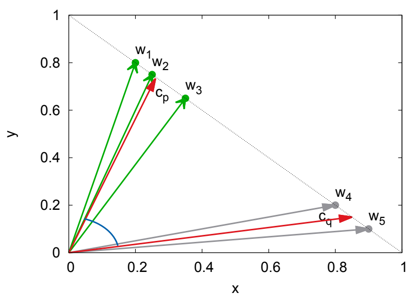


Figure 1: Example of dissimilarity function.

Definition 4. ***(Dissimilarity function*** f_d***).*** *Given a set of data objects* S*, a set of weighting vectors* W*, two objects* $p, q \in S$*, and their respective centroids* $\mathbf{c}_p$ *and* $\mathbf{c}_q$*, the distance of* p *and* q *is defined based on the cosine similarity of the centroids:*

$$f_d(RTOP_k(p), RTOP_k(q)) = 1 - cos(\mathbf{c}_p, \mathbf{c}_q)$$

The advantage of using the centroid $\mathbf{c}_p$ instead of the actual set of vectors $RTOP_k(p)$ is that the centroid is a compact and accurate representation of the set, which in turn allows efficient processing of the dissimilarity function, compared to other dissimilarity metrics that operate on sets of arbitrary size. As a result, we use $d(p, q) = 1 - cos(\mathbf{c}_p, \mathbf{c}_q)$ as distance function in this paper[1].

Example 1. *Figure 1 shows an example of the reverse top-k sets* $RTOP_k(p) = \{\mathbf{w}_1, \mathbf{w}_2, \mathbf{w}_3\}$ *and* $RTOP_k(q) = \{\mathbf{w}_4, \mathbf{w}_5\}$*, which belong to products p and q respectively. In the Euclidean space, a linear top-k query can be represented by a vector* $\mathbf{w}$ *[20, 21]. The magnitude of the query vector does not influence the query result, as long as the direction remains the same, therefore without loss of generality we assume that* $\sum_{i=1}^{m} w[i] = 1$*. In the 2-dimensional space, all weighting vectors belong to the line as depicted in Figure 1. Moreover, top-k queries defined by similar weighting vectors* $\mathbf{w}$ *are expected to produce similar result sets [20, 21]. Thus, the weighting vectors of the reverse top-k set of p are expected to lie nearby on the line. Furthermore, for a hypothetical weighting vector which lies on the line between* $\mathbf{w}_1$ *and* $\mathbf{w}_3$*, it is expected that p is highly ranked, and therefore it is highly probable that this vector would belong to the reverse top-k set of p.*

The centroid of the weighting vectors captures the above intuitions, and the angle between two centroids represents the dissimilarity of the weighting vectors. Obviously, different functions for set dissimilarity (hence also for measuring distance) are supported by our approach, including (for instance) the Jaccard similarity of the reverse top-k sets. Nevertheless, the Jaccard similarity fails to capture the locality of the weighting vectors.

Furthermore, we define the *diversity* $div(D)$ of a set of objects $D \subseteq S$. Notice that the set D^* with the highest diversity value $div(D^*)$ among all r-sets of points in S, is the optimal solution for Problem 1. The diversity value $div(D)$ is normalized in [0,1].

[1]In a slight abuse of notation, we also use $d(\mathbf{u}, \mathbf{v}) = 1 - cos(\mathbf{u}, \mathbf{v})$ to denote the cosine distance between any two vectors $\mathbf{u}$ and $\mathbf{w}$.

Definition 5. ***(Diversity value)*** *Given a set of points* S*, a subset* $D \subseteq S$ *of size* r*, and set of vectors* W*, we define as* diversity *of* D*:*

$$div(D) = \frac{2}{r(r-1)} \sum_{\substack{p,q \in D \\ p \neq q}} (1 - cos(\mathbf{c}_p, \mathbf{c}_q))$$

5. ALGORITHMS WITH CENTROID COMPUTATION

The process of discovering r diverse products $D_r(S)$ from a set of products S consists of two main steps: (1) identifying a set C of *candidate centroids* that correspond to candidate products for inclusion in the most diverse products (Section 5.1), and (2) selecting r of these candidates as the most diverse products (Section 5.2).

Each candidate centroid in $\mathbf{c}_p \in C$ corresponds to one product $p \in S$ and is the centroid vector of the $RTOP_k(p)$ set of p. More formally $C = \{\mathbf{c}_p | p \in S, RTOP_k(p) \neq \emptyset, \mathbf{c}_p \text{ is centroid of } RTOP_k(p)\}$. Obviously, products that are not preferable for any customer are ignored.

Algorithm 1 describes the afore-described method and returns a set of r diverse products. In line 1, the candidate centroids C are computed using any of the methods that will be described in Section 5.1. As the set of centroids C may be large depending on the data distribution, a sample R of fixed size s is created by picking centroids uniformly at random (line 2). Finally, in line 3, the second step entails solving the r-Diversity problem by applying a greedy algorithm, called Diverse Product Selection Algorithm (*DPSA*), on the sampled set of centroids R, as will be described in Section 5.2.

Algorithm 1 r-Diverse Products

```
Input: $S$: set of products
       $W$: set of weighting vectors
       $k$: value of top-$k$ and reverse top-$k$
       $s$ : size of initial sample
       $r$ : required number of diverse products
Output: $D_r(S)$: the set of $r$ most diverse products of $S$

1: $C \leftarrow$ CandidateCentroids($S$, $W$, $k$)
2: $R \leftarrow$random subset of $C$ with $|R| = s$
3: $D_r(S) \leftarrow DPSA(C, R, r)$
4: return $D_r(S)$
```

5.1 Retrieving the Candidate Centroids

Different alternatives exist in order to compute the set C of candidate centroids. In the following, we present three alternative methods for determining the set C. Notice that all methods produce an identical set C of centroids.

The most straightforward method is to perform a reverse top-k query for each product p in S and compute the centroid vector of each set $RTOP_k(p)$ using Definition 3. We denote this approach *Rtopk*. Its processing cost is basically determined by the computation of $|S|$ reverse top-k queries. Since any existing algorithm for reverse top-k processing can be employed for the underlying reverse top-k computation, this method is quite generic.

An improvement of the first method is derived based on the observation that some products have empty reverse top-k sets (i.e., they do not belong to the top-k result of any weighting vector). Hence, it is possible to avoid processing some reverse top-k sets. To achieve this, we exploit the progressive result generation of the algorithm in [22], which is able to retrieve objects in decreasing order of the sizes of their reverse top-k sets. We denote this method as *Itopk* based on the fact that the algorithm [22] has been proposed for retrieval of influential objects. As a result, we avoid processing a reverse top-k query for objects with empty reverse top-k sets, thus improving the performance of *Rtopk*.

The third method exploits the observation that it may be more efficient to process all top-k queries, instead of processing multiple reverse top-k queries. Thus, we perform a top-k query for each preference vector $w \in W$, which makes straightforward the computation of the reverse top-k sets of any data object, and hence also their respective centroids. In fact, the top-k sets do not need to be maintained until all top-k queries have been processed, but instead the centroids can be calculated progressively. For each retrieved object in the top-k result set, the centroid is updated by adding the new vector $\mathbf{w}$ to its previous centroid, while also the number of vectors per object is maintained. After finishing all top-k queries, for each centroid the coordinates are divided by the cardinality of the reverse top-k set. Since top-k queries for all vectors in W are processed, we call this method all top-k, i.e., *Atopk*. An advantage of *Atopk* is that the processing cost in terms of top-k evaluations is fixed, namely $|W|$ top-k queries, in contrast to *Rtopk* and *Itopk* where in the worst case the top-k evaluations can be up to $|W| \cdot |S|$. Thus, the efficiency of *Atopk* is influenced slightly by the cardinality of S, in contrast to *Rtopk* which computes the reverse top-k set even for products with empty reverse top-k sets.

5.2 Diverse Product Selection Algorithm

After having computed the centroid vectors of all non-empty reverse top-k sets, the next step is to find the r most diverse centroids and the products that they represent. As already mentioned, the r-Diversity problem is defined as a dispersion problem that is known to be NP-hard [8]. Thus, computing the optimal solution for the r-Diversity problem is not feasible even for relatively small data sets. Hence, we employ an algorithm that efficiently computes an approximate solution of high quality [5]. More specifically, we use a greedy algorithm, called Diverse Product Selection Algorithm (*DPSA*), that iteratively selects the next centroid that maximizes the value of the objective function. Its pseudocode is depicted in Algorithm 2.

Description. The algorithm takes as input the set of candidate centroids C, a random sample set R of the candidate centroids that is going to be used, and an integer r which is the desired number of most diverse products. It returns an approximate set $D_r(S)$ of the r most diverse products and their centroids. The sample R is typically much smaller in size than C, in order to reduce the cost of the first part of the algorithm, which is to find the two most distant vectors in R (line 2) and add them to the result set $D_r(S)$ (line 3). Then, the algorithm iteratively selects the next centroid $\mathbf{c}_q$ until r centroids have been retrieved (loop in line 4). Each time, the selected centroid is the one that maximizes the sum of distances from the already selected most diverse vectors $D_r(S)$. Notice that R is used only for the initialization of $D_r(S)$ (line 3), while the remaing centroids are selected from C.

Complexity. The selection of the two most diverse products (seeds) has a cost $O(|R|^2) = O(s^2)$. The remaining part of the algorithm has a cost of $O(r^2|C|)$ and therefore the total cost is equal to $O(s^2 + r^2|C|)$. If no sample is used ($s = |C|$) in the seed selection then the algorithm will have a cost of $O(|C|^2)$.

Algorithm 2 Diverse Product Selection Algorithm *DPSA*()

```
Input: $C$ : set of candidate centroids
       $R$ : sample of $C$
       $r$ : required number of diverse products
Output: $D_r(S)$: the set of $r$ most diverse products of $S$

1: $\mathbf{c}_{p1}, \mathbf{c}_{p2} \leftarrow \mathbf{c}_{p1}, \mathbf{c}_{p2} : \forall \mathbf{c}_{pi}, \mathbf{c}_{pj} \in R : d(\mathbf{c}_{p1}, \mathbf{c}_{p2}) \geq d(\mathbf{c}_{pi}, \mathbf{c}_{pj})$
2: $C \leftarrow C - \{p_1, p_2\}$
3: $D_r(S) \leftarrow \{p_1, p_2\}$
4: while $|D_r(S)| < r$ do
5:     $\mathbf{c}_q = \arg\max_{\mathbf{c}'_q \in C} \left(\sum_{p \in D_r(S)} d(\mathbf{c}'_q, \mathbf{c}_p)\right)$
6:     $D_r(S) \leftarrow D_r(S) \bigcup \{q\}$
7:     $C \leftarrow C - \{q\}$
8: end while
9: return $D_r(S)$
```

Implementation details. In each loop iteration of the *DPSA* algorithm (lines 4-8), the algorithm calculates the sum of distances between a centroid vector $\mathbf{c_q} \in C - D_r(S)$ and the centroid vectors in $D_r(S)$. As described above this procedure has a cost of $O(r^2|C|)$. In the case of the cosine distance we can reduce this cost to $O(r|C|)$ by exploiting the properties of the cosine function. As shown in Equation 1 the sum of distances of $\mathbf{c}_q$ to all centroids in $D_r(S)$ is equal to $|D_{r(S)}| - \frac{\mathbf{c}'_q}{|\mathbf{c}'_q|} \cdot \mathbf{c}_{D_r(S)}$. In that way it is only necessary to calculate the centroid of $D_r(S)$ before each loop iteration.

$$\begin{aligned}\sum_{p \in D_r(S)} d(\mathbf{c}'_q, \mathbf{c}_p) &= \sum_{p \in D_r(S)} 1 - cos(\mathbf{c}'_q, \mathbf{c}_p) \\ &= |D_r(S)| - \frac{\mathbf{c}'_q}{|\mathbf{c}'_q|} \cdot \sum_{p \in D_r(S)} \frac{\mathbf{c}_p}{|\mathbf{c}_p|} \qquad (1) \\ &= |D_r(S)| - \frac{\mathbf{c}'_q}{|\mathbf{c}'_q|} \cdot \mathbf{c}_{D_r(S)}\end{aligned}$$

6. SELECTIVE TOP-K ALGORITHM

The main drawback of the previous algorithm is that it requires the computation of all centroids, which has a significant processing cost regardless of the employed method for candidate centroid computation. In particular, depending on the cardinality of W and S, the computation of the centroids may be time-consuming. In order to alleviate this shortcoming, in this section, we propose a method that fuses the centroid computation with the selection of diverse objects. Our goal is to efficiently compute an approximation of the centroids (by evaluating only a handful of carefully selected top-k queries), which is sufficient to produce a set of r products with high diversity.

6.1 Centroid Approximation

Conceptually, the proposed algorithm uses a series of iterations, where each iteration consists of three parts: (1) select a weighting vector $\mathbf{w}_i$ in order to process the top-k query it defines, (2) compute an approximation of the centroid-set based on the results of all already processed top-k queries, and (3) select diverse products by invoking the *DPSA* algorithm (Section 5.2) with input the approximate centroid-set. In each iteration, a top-k query based on $\mathbf{w}_i$ is executed. Some objects $p \in TOP_k(\mathbf{w}_i)$ may not have been retrieved before and those are added to the centroid-set. For the remaining objects $p \in TOP_k(\mathbf{w}_i)$ the approximate centroid is updated, since $\mathbf{w}_i$ is added to their reverse top-k sets. In fact, the reverse top-k sets are not maintained, but the centroid of an object is computed progressively as in the case of *Atopk*. Thus, in each iteration the centroid-set is only an approximation of the candidate centroids C computed by Algorithm 1 because (a) C may contain more centroids as some objects may not have been retrieved yet and (b) the centroids of an object p are estimated based on a limited set of top-k queries only. However, in each iteration, the candidate-set is enriched with the results of the next top-k query. Additionally, a set of r diverse products $D_r(S)$ is computed based on the current set of centroids. Finally, the selection of the next weighting vector to be processed is based on maximizing the sum of distances to the set of centroids defined by $D_r(S)$.

The main idea of our algorithm is that the maximum cosine distance (i.e., maximum diversity) of two objects is bounded by the maximum cosine distance of any two weighting vectors. Let us assume that $\mathbf{w}_1$ and $\mathbf{w}_2$ are the two weighting vectors with the maximum cosine distance (the most diverse). Let us further assume that there exist two products p_1 and p_2 for which holds: $RTOP_k(p_1) = \{\mathbf{w}_1\}$ and $RTOP_k(p_2) = \{\mathbf{w}_2\}$. Then, it holds that for 2-Diversity problem the optimal solution is $\{p_1, p_2\}$ and their diversity equals to $1 - cos(\mathbf{w}_1, \mathbf{w}_2)$, since $\mathbf{c}_{p_i} = \mathbf{w}_i$. If more weighting vectors belong to $RTOP_k(p_1)$ then the diversity between $\{p_1, p_2\}$ decreases. Therefore, our algorithm starts by evaluating the top-k queries for the most diverse weighting vectors. In each step, the most diverse weighting vector to the current most diverse centroids is selected, as each centroid may summarize several weighting vectors.

6.2 Algorithmic Description

Algorithm 3 shows the pseudocode of the proposed algorithm that uses a limited set of top-k queries only. We call this algorithm *Selective Top-k Algorithm* and denote it with *Stopk*.

Description. The first major difference to Algorithm 1 is that the initial centroid computation is avoided. First, the algorithm computes a random sample W' (of size s) of W (line 1) and the two most dissimilar weighting vectors $\mathbf{w}_1$ and $\mathbf{w}_2$ of W' are selected (line 2). Notice that the sample W' is produced uniformly at random, thus it follows the distribution of W and is used only for the initialization of $\widehat{\mathcal{C}}$. Applying the initialization step on W would result in a cost of $O(|W|^2)$, while with the sample it is reduced to $O(|W'|^2)$. Next, the top-k queries for $\mathbf{w}_1$ and $\mathbf{w}_2$ are processed and from the resulting merged set of products a set $\widehat{\mathcal{C}}$ of centroids is computed (line 3). Notice that $\widehat{\mathcal{C}}$ is computed based solely on the products retrieved thus far by the two top-k queries. These centroids form the candidate set for finding the most diverse products. In the following step, the Algorithm 2 is invoked with input the candidate set, and the two most diverse products are retrieved and placed in $D_r(S)$ (line 4). Note that $\widehat{\mathcal{C}}$ is much smaller than C, thus *DPSA* algorithm is applied on $\widehat{\mathcal{C}}$ and no random sample is required.

Algorithm 3 Selective Top-k Algorithm

```
Input: S: set of products
       W: set of weighting vectors
       k: value of top-k and reverse top-k
       s : size of initial sample
       r : required number of diverse products
       steps : number of iterations (steps ≥ r)
Output: D_r(S): the set of r most diverse products of S

 1: $W' \leftarrow$random subset of $W$ with $|W'| = s$
 2: $\mathbf{w}_1, \mathbf{w}_2 \leftarrow \mathbf{w}_1, \mathbf{w}_2 : \forall \mathbf{w}_i, \mathbf{w}_j \in W' : d(\mathbf{w}_1, \mathbf{w}_2) \geq d(\mathbf{w}_i, \mathbf{w}_j)$
 3: $\widehat{\mathcal{C}} \leftarrow$ ComputeCentroids($\bigcup_{x=1,2} TOP_k(\mathbf{w}_x)$)
 4: $D_r(S) \leftarrow DPSA(\widehat{\mathcal{C}}, \widehat{\mathcal{C}}, 2)$
 5: $i = 2$
 6: while $i < steps$ do
 7:     $i{+}{+}$
 8:     $\mathbf{w}_i = \arg\max_{\mathbf{w} \in W} \left( \sum_{p \in D_r(S)} d(\mathbf{c}_p, \mathbf{w}) \right)$
 9:     $\widehat{\mathcal{C}} \leftarrow$ ComputeCentroids($\bigcup_{x=1 \ldots i} TOP_k(\mathbf{w}_x)$)
10:     $D_r(S) \leftarrow DPSA(\widehat{\mathcal{C}}, \widehat{\mathcal{C}}, min(i+1, r))$
11: end while
12: return $D_r(S)$
```

In each iteration, the most dissimilar weighting vector $\mathbf{w}_i$ to the centroid vectors $\mathbf{c}_p$ ($p \in D_r(S)$) is selected (line 8). For this $\mathbf{w}_i$, the respective top-k query is executed and the candidate list $\widehat{\mathcal{C}}$ is updated as before (line 9). Then, the *DPSA* algorithm is invoked again to produce a new set of diverse products (line 10). The same procedure is repeated for at least r steps. Notice that when the iteration counter i is smaller than r, the algorithm produces i diverse products, and only when i becomes greater than r does the algorithm return r diverse products.

In order to improve further the approximation of the centroids more iterations can be applied. The number of iterations (*steps*) is a system parameter that captures an interesting trade-off between the diversity of the result set and the processing time. Small values of *steps* increase the efficiency of the algorithm by reducing its processing time. In the experimental evaluation, we demonstrate that a small number of iterations is sufficient to produce results with diversity comparable to that of Algorithm 1, with significantly lower processing cost. Notice that in the extreme case that the number of iterations of *Stopk* is set equal to the cardinality of W and also no sampling is used ($s = |W|$), the set of diverse products and number of required top-k queries will be the identical with *Atopk*. Nevertheless, in this case, *Stopk* will have the computational overhead of applying multiple times the *DPSA* algorithm and finding the diverse weighting vectors.

EXAMPLE 2. *Table 3 shows an example of the execution of Stopk for $r = 2$. We assume that in the initialization step vectors $\mathbf{w}_1$ and $\mathbf{w}_2$ are selected. Furthermore (assuming $k = 3$), the top-3 results for the selected vectors are depicted. After the initialization step, the sets of approximate centroids $\widehat{\mathcal{C}}$ contains 5 centroids (namely $\mathbf{c}_{p_1}$, ..., $\mathbf{c}_{p_5}$),*

Initialization step:
$TOP_k(\mathbf{w}_1) = p_1, p_2, p_3$, $TOP_k(\mathbf{w}_2) = p_2, p_4, p_5$
$\mathbf{c}_{p_1} = \mathbf{w}_1$, $\mathbf{c}_{p_2} = \frac{1}{2}(\mathbf{w}_1 + \mathbf{w}_2)$,
$\mathbf{c}_{p_3} = \mathbf{w}_1$, $\mathbf{c}_{p_4} = \mathbf{w}_2$, $\mathbf{c}_{p_5} = \mathbf{w}_2$
First step:
$TOP_k(\mathbf{w}_3) = p_3, p_4, p_5$
$\mathbf{c}_{p_1} = \mathbf{w}_1$, $\mathbf{c}_{p_2} = \frac{1}{2}(\mathbf{w}_1 + \mathbf{w}_2)$, $\mathbf{c}_{p_3} = \frac{1}{2}(\mathbf{w}_1 + \mathbf{w}_3)$
$\mathbf{c}_{p_4} = \frac{1}{2}(\mathbf{w}_2 + \mathbf{w}_3)$, $\mathbf{c}_{p_5} = \frac{1}{2}(\mathbf{w}_2 + \mathbf{w}_3)$
Second step:
$TOP_k(\mathbf{w}_4) = p_1, p_2, p_6$
$\mathbf{c}_{p_1} = \frac{1}{2}(\mathbf{w}_1 + \mathbf{w}_4)$, $\mathbf{c}_{p_2} = \frac{1}{3}(\mathbf{w}_1 + \mathbf{w}_2 + \mathbf{w}_4)$,
$\mathbf{c}_{p_3} = \frac{1}{2}(\mathbf{w}_1 + \mathbf{w}_3)$, $\mathbf{c}_{p_4} = \frac{1}{2}(\mathbf{w}_2 + \mathbf{w}_3)$,
$\mathbf{c}_{p_5} = \frac{1}{2}(\mathbf{w}_2 + \mathbf{w}_3)$, $\mathbf{c}_{p_6} = \mathbf{w}_4$

Table 3: Example of *Stopk*.

which correspond to the data points that have been retrieved by at least one top-k query. Algorithm 2 is applied on $\widehat{C}$ and we assume that $\mathbf{c}_{p_1}$ and $\mathbf{c}_{p_4}$ as the two most diverse vectors. In the first iteration of Stopk, the most diverse (according to $\mathbf{c}_{p_1}$ and $\mathbf{c}_{p_4}$) weighting vector of W is selected. In this step, $\mathbf{w}_3$ is selected and the approximate centroids $\widehat{C}$ are updated based on $TOP_k(\mathbf{w}_3)$ as depicted in Table 3. Again, Algorithm 2 is applied on $\widehat{C}$ and $\mathbf{c}_{p_1}$ and $\mathbf{c}_{p_4}$ are identified as the two most diverse vectors. Stopk continues with a second iteration by evaluating $\mathbf{w}_4$. In this step, $\mathbf{c}_{p_6}$ is added to $\widehat{C}$ as it belongs to $TOP_k(\mathbf{w}_4)$. Again Algorithm 2 will be invoked and the most dissimilar weighting vector of W will be selected. The same procedure continues until steps iterations has been executed.

Complexity. To perform a cost analysis of the algorithm, we need to identify its basic cost factors. These factors include the initial computation of the two most dissimilar vectors ($O(s^2)$), the processing cost of *steps* top-k queries, the cost of determining the next most dissimilar weighting vector ($O(steps \cdot r \cdot |W|)$) (line 8), and the cost induced by invoking the *DPSA* algorithm *steps* times. The cost of processing *steps* top-k queries will always be much cheaper than Algorithm 1, which needs to process W top-k queries (in the case of *Atopk*) to perform the centroid computation. It should also be noted that the calls to the *DPSA* algorithm are much cheaper, because it operates on $\widehat{C}$ which is much smaller than C. Overall, the cost of the algorithm is $s^2 + steps \cdot r \cdot |W|$, since these are the dominant cost factors, and the complexity is linear with respect to W ($O(steps \cdot r \cdot |W|)$), when *steps* is small (r is typically small anyway).

7. MAINTENANCE

In this section, we present techniques for maintaining the diverse set of products in the case of dynamic data. In fact, the methodology of *Stopk* (Algorithm 3) can be applied to maintain the r-diverse products. We consider two cases: (1) new products are inserted in the product database, and (2) new preferences (in the form of weighting vectors) are added in the customer preference database. Both cases actually occur in online shops, when new products appear in the market and new customer preferences are extracted from social sites.

In order to support product insertions efficiently, we exploit the top-k queries that where computed during the computation of $\widehat{C}$. Let W^* be the set of weighting vectors for which the top-k queries have been computed. We maintain for each weighting vector $\mathbf{w} \in W^*$ the score of the k-th product. When a new product p_{new} is inserted in the database, we check for each query $\mathbf{w} \in W^*$ if p_{new} has a better score than the k-th product. If this does not occur for any $\mathbf{w} \in W^*$, we can safely ignore p_{new}, as it does not affect the determination of the diverse products. On the other hand, if p_{new} becomes top-k for some weighting vector $\mathbf{w}$, we compute the new centroids only for the affected products (i.e., p_{new} and the products p_i that used to be at the k-th rank, but were evicted by p_{new}) and update the set $\widehat{C}$. We then apply *DPSA* algorithm on $\widehat{C}$ and produce a new set of diverse products. Note that in the first case, we can ensure that the result set is the same as in the case where *Stopk* would be executed on the updated data set, but this does not hold for the second case. The similarity of the centroids before and after the update can be used in order to decide when the *Stopk* algorithm should be invoked to have a result set of higher quality.

In order to be able to handle new preferences effectively, during the computation of the diverse product we maintain the minimum (*min*) of all maximal sums of distances between a centroid and any selected weighting vector (i.e., the expression in line 8 of Algorithm 3). In the case of a new weighting vector $\mathbf{w}_{new}$, we follow the same principle as Algorithm 3 to decide whether the respective top-k query should be evaluated. If $\sum_{p \in D_r(S)} d(\mathbf{c}_p, \mathbf{w}_{new})$ is larger than *min*, then the top-k query for $\mathbf{w}_{new}$ is computed, the set of centroids $\widehat{C}$ updated and *DPSA* algorithm is executed on $\widehat{C}$ to produce a new set of diverse products. Intuitively, when vector $\mathbf{w}_{new}$ induces small changes to the set of centroids, we do not need to recompute the r-diverse products as $\mathbf{w}_{new}$ would not have been selected by *Stopk* in any case. Again, the retrieved diverse products are not the same as if *Stopk* would be executed on the updated weighting vector set, therefore a threshold on the similarity of the centroids before and after the update may trigger executing *Stopk* to get a set of higher quality.

8. SUPPORTING OTHER SET-BASED SIMILARITY FUNCTIONS

In general, our approach is applicable also for other functions that compute the similarity between sets of vectors. In such a case, our algorithms would not calculate centroids (which is simply a representation of a set of weighting vectors), but would instead directly operate on the reverse top-k sets of products. Following this line of thought, $\hat{C}$ would represent a set of approximate reverse top-k sets (instead of a set of centroids) and the computation of the most diverse sets becomes independent of the similarity or distance function.

In more technical terms, Algorithm 1 would not calculate centroids but would only maintain the reverse top-k sets, and Algorithm 3 would not compute centroids incrementally but would simply update the approximate reverse top-k sets of products based on the executed top-k queries. Then, Algorithm 2 can be directly applied to the reverse top-k sets.

For instance, one popular similarity function is the Jac-

Parameter	Values	
Datasets	UN, CO, AC, CL NBA, HOUSE	
Data cardinality	1K, **5K**, 10K	(Diversity Quality)
	5K, **10K**, 30K	(Scalability Analysis)
	100K, 200K, 500K	(Sensitivity Analysis)
	17265, 127930	(Real Datasets)
Weight cardinality	1K, **5K**, 10K	(Diversity Quality)
	5K, **10K**, 30K	(Scalability Analysis)
	100K, 200K, 500K	(Sensitivity Analysis)
	100K, 200K, 500K	(Real Datasets)
# results(r)	3, 4, **5**	(Diversity Quality)
	10	(Scalability Analysis)
	5, **10**, 30	(Sensitivity Analysis)
	5, **10**, 30	(Real Datasets)
# top-k results(k)	**10**, 20	(Diversity Quality)
	5, **10**, 30, 50	(Scalability Analysis)
	5, 10, **30**, 50	(Sensitivity Analysis)
	5, 10, **30**, 50	(Real Datasets)
# dimensions(m)	3	(Diversity Quality)
	3	(Scalability Analysis)
	3, 4, 5, 6	(Sensitivity Analysis)

Table 4: Parameters (default values in bold).

card similarity, which is defined as the size of the intersection divided by the size of the union of two sets. Our approach readily supports the Jaccard similarity on reverse top-k sets, as outlined above. Notice that the advantage of the cosine similarity compared to Jaccard for the problem of finding diverse products is that it returns more fine-grained similarity values. For example, in the case of disjoint sets, the Jaccard similarity value equals to zero, and does not distinguish between the sets. Instead, the cosine similarity of the centroid vectors allows us to distinguish between them by returning a non-zero value. Moreover in the case of a set A that is a subset of another set B ($A \subseteq B$), the Jaccard similarity is equal to $\frac{|B|-|A|}{|B|}$ which can get arbitrarily close to the maximum value. In such cases, using the Jaccard similarity would not be helpful, as it could lead to selecting products which are possibly covered by others. The centroid vectors reduce this problem (they do not eliminate it) by choosing sets that are selected by distant user preferences.

9. EXPERIMENTAL EVALUATION

In this section, we present the results of the experimental evaluation. All algorithms were implemented in Java and the experiments run on 2x Intel Xeon X5650 Processors (2.66GHz). The algorithms are disk-based and the index structure used was an R-tree with a buffer size of 100 blocks and the block size is 4KB. The main parameters and values used through the experiments are presented in Table 4.

Data sets. For the data set S, we use both synthetic and real data. We examine four different synthetic data distributions, namely uniform (UN), correlated (CO), anticorrelated (AC) and clustered (CL). For the uniform data set, the data object values for all m dimensions are generated independently using a uniform distribution. The correlated and anticorrelated data sets are generated as described in [3]. The clustered data set was created as follows: first 5 cluster centroids were selected randomly. Then, each coordinate is generated on the m-dimensional space by following a normal distribution on each axis with variance $\sigma_S^2 = 0.345$, and a mean equal to the corresponding coordinate of the centroid. In addition, we use a real data set. HOUSE (Household) consists of 127930 6-dimensional tuples, representing the percentage of an American family's annual income spent on 6 types of expenditure: gas, electricity, water, heating, insurance, and property tax. For the data set W of the weighting vectors, we used a uniform (UN) distribution.

Algorithms. We implemented the three algorithms for centroid computation (*Rtopk*, *Itopk*, and *Atopk*) coupled with the *DPSA* algorithm as described in Section 5, and the selective top-k algorithm (*Stopk*) described in Section 6. We also implemented an exact algorithm (denoted *opt*) that finds the optimal solution, but obviously cannot scale well. For reverse top-k processing, *Rtopk* uses the state-of-the-art BBR* algorithm [23], while *Itopk* uses the BB algorithm [22]. In all algorithms the data set is indexed by an R-tree and for the top-k query processing we employ a state-of-the-art branch-and-bound algorithm [17].

Metrics. The metrics under which we evaluated the implemented algorithms were: (a) execution time required by each algorithm, (b) I/O accesses, (c) achieved diversity values. We also measured the number of processed top-k queries, but in the interest of space we do not report them since they follow exactly the I/O metric. We measure only the I/O induced on the data set S, since the I/O on W are caused by sequential access and accessing data sequentially is much faster than the random accesses of S.

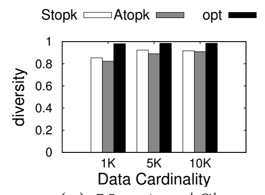
We conduct an experimental study varying the parameters of dimensionality (3-6), cardinality (1K-500K) of S, cardinality (1K-500K) of W, value of k (5-50), value of r (3-30), sample size $|W'|$ ($0.001|W|$-$0.1|W|$), and number of *steps* (100-1000). Each experiment was repeated 10 times over different instances of the data sets with the same parameters and different seed to the random generator, in order to factor out the effect of randomization. Average values are reported in all cases.

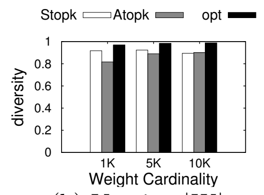
Evaluation methodology. Our evaluation was divided in three parts. In the first part (9.1), we compare the algorithms *Atopk* and *Stopk* against the exact algorithm (*opt*) in order to evaluate the quality of approximation of diversity. In the second part (9.2), we evaluate the performance of *Stopk* against the algorithms that rely on centroid computation (*Rtopk*, *Itopk*, and *Atopk*). In the last part (9.3 and 9.4), we perform a thorough sensitivity analysis of *Atopk* (which proved to perform best among the algorithms with centroid computation) against *Stopk*. We should stress that the diversity of the result set of *Stopk* is calculated using the whole set of preferences W, and not only the vectors used for the identification of the most diverse products.

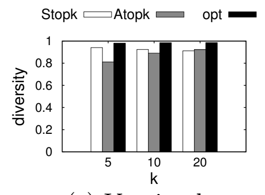
9.1 Quality of Diversity

The purpose of this series of experiments is to study the loss of diversity compared to optimal solution when using our algorithms *Atopk* and *Stopk*. Thus, we compare to the optimal diversity produced by an exact algorithm (*opt*), which examines all possible $\binom{|S|}{r}$ combinations of products exhaustively to find the optimal solution. Recall that *Rtopk* and *Itopk* produce exactly the same result set as *Atopk* and therefore have also the same diversity. The default setup for this series of experiments was: m=3, $|S| = 5K$, $|W| = 5K$, $k = 10$, $r = 5$, $s = 0.1 \cdot |W|$, $steps = 100$, S:UN, and W:UN.

Figure 2 depicts the diversity values for varying different parameters, namely $|S|$, $|W|$, r and k. The diversity achieved by the greedy algorithm (*Atopk*) is in most cases quite close to optimal, while *Stopk* results in similar diver-


(a) Varying $|S|$


(b) Varying $|W|$


(c) Varying k

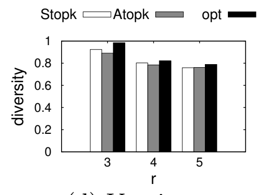
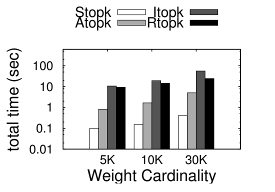

(d) Varying r

Figure 2: Comparison to optimal diversity value.


(a) Time

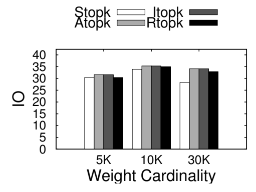

(b) I/O

Figure 3: Performance when varying $|W|$.

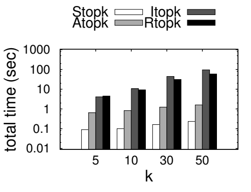
sity values. As we can observe, our approximate algorithms perform very well in comparison with the exact algorithm. In the worst case, when the size of the objects data set is only $|S|$=1000 objects (Figure 2(a)), the maximum difference in diversity is 19%. As the datasets grow in size, the diversity of the result set of the approximate algorithms approaches the optimal diversity. It is noteworthy that as the number of returned objects (r) increases, the diversity value drops. This behavior is expected as the more points we select the smaller their average distance will become.

We omit the figures comparing the performance of our algorithms to *opt*, since, as expected, our algorithms outperformed the exact approach by 1-4 orders of magnitude in terms of execution time.

9.2 Scalability Analysis

In this series of experiments we compare the performance of the algorithms with centroid computation (described in Section 5) in terms of execution time and I/O. We also include the *Stopk* algorithm in the charts for completeness. The default setup for this series of experiments is m=3, $|S| = 10K$, $|W| = 10K$, $k = 10$, $r = 10$, S:UN, W:UN, $steps = 100$, $s = 0.01 \cdot |W|$.

Varying weight cardinality $|W|$. As Figure 3 illustrates, *Atopk* and *Stopk* outperform by orders of magnitude the *Rtopk* and *Itopk* algorithms in terms of execution time. This difference is not reflected in the measured I/O, because of the use of the buffer of the R-tree. When the number of issued top-k queries is considered, both *Rtopk* and *Itopk* process at least one order of magnitude more top-k queries than *Atopk* and *Stopk*. This processing cost is responsible for their slow runtime. We note that even though both *Rtopk* and *Itopk* are more efficient than *Atopk* when a single


(a) Time

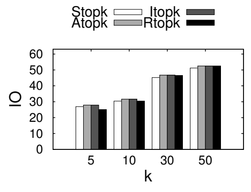

(b) I/O

Figure 4: Performance when varying k.

reverse top-k query or a small number of influential points is needed, they are less efficient when they are run repeatedly multiple times. In this case, *Atopk* has a benefit and performs better. In particular, *Rtopk* has no memory of the completed executions for different queries, and therefore it computes repeatedly the top-k results of many preference vectors. On the other hand, *Itopk* performs fewer reverse top-k queries than *Rtopk*, but shares the same shortcoming for those reverse top-k queries that it processes. Therefore, it faces the same problem as the *Rtopk*, however in not such great extent. Similar conclusions are drawn when varying the data cardinality $|S|$. The performance of the algorithms is much less affected by the increase of data cardinality, as during the execution of the top-k queries very few data objects are accessed.

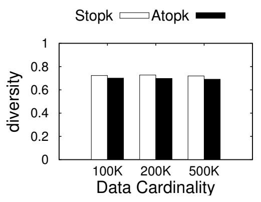
Varying k. Figure 4 illustrates the effect of varying parameter k. *Atopk* consistently outperforms *Rtopk* and *Itopk* in terms of time, while *Stopk* improves further the performance in terms of both time and I/O. *Atopk*, *Rtopk* and *Itopk* have the same performance in terms of I/O due to the R-tree buffer employed during query processing. Furthermore, for all algorithms, both time and I/O increase for increasing values of k.

Since *Atopk* consistently outperforms the two other algorithms (*Rtopk* and *Itopk*) that rely on centroid computation, we use only *Atopk* in the remaining experiments as representative of this family of algorithms.

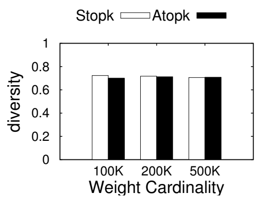
9.3 Sensitivity Analysis

In this section, we provide a detailed sensitivity analysis by varying different parameters that influence the performance of our proposed algorithms. Due to space limitations, for some setups we omit the figures depicting the I/O since the time indicates the efficiency of the algorithms. The default setup for this series of experiments is m=3, $|S| = 100K$, $|W| = 100K$, $k = 30$, $r = 10$, S:UN, W:UN, $steps = 500$, $s = 0.01 \cdot |W|$.

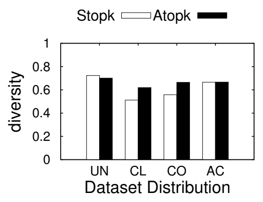
Sensitivity analysis for varying $|S|$, $|W|$, Data Distribution and m. In Figure 5, we study the behavior of *Atopk* and *Stopk* for increasing cardinality of the data set S and the weighting vectors W, as well as for various data distributions (UN,CL,CO,AC) and dimensionality values m. First, we examine how the diversity values of *Atopk* and *Stopk* for the different parameters are influenced (Figures 5(a)–5(d)). We observe that *Stopk* benefits from the increased size of the data set and the preferences set. *Stopk* retrieves a set of r objects that have similar diversity compared to *Atopk*. In most cases the diversity achieved by *Stopk* is almost equal to the one achieved by *Atopk* and in some cases it is even slightly higher. This happens because *Stopk* locates the most diverse preferences and based



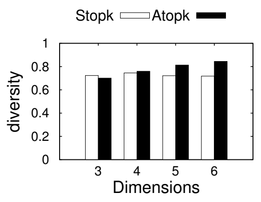
(a) Diversity: Varying $|S|$



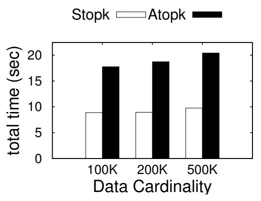
(b) Diversity: Varying $|W|$



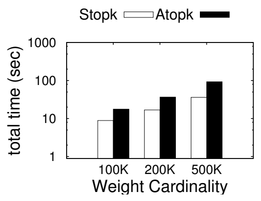
(c) Diversity: Data distribution



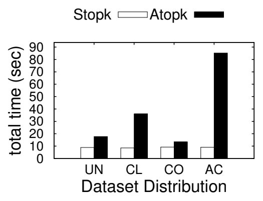
(d) Diversity: Varying m



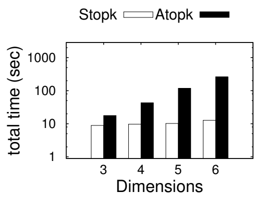
(e) Time: Varying $|S|$



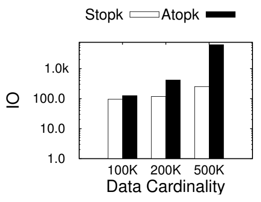
(f) Time: Varying $|W|$



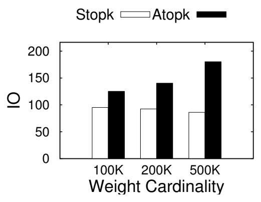
(g) Time: Data distribution



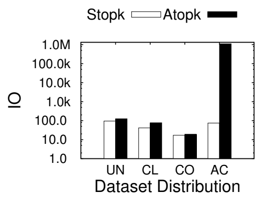
(h) Time: Varying m



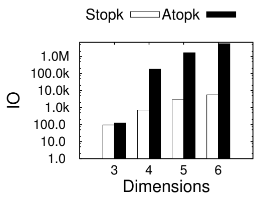
(i) I/O: Varying $|S|$



(j) I/O: Varying $|W|$



(k) I/O: Data distribution



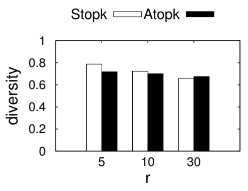
(l) I/O: Varying m

Figure 5: Sensitivity analysis for varying $|S|$, $|W|$, data distribution and m.

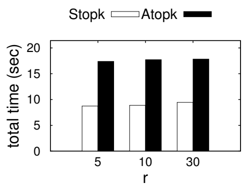
on them it identifies the most diverse products. *Atopk* on the other hand, bases the search for most diverse objects on the centroids of the $RTOP_k$ sets of the products. For increased dimensionality, as depicted in Figure 5(d), the diversity value of *Stopk* compared to *Atopk* is influenced more than for the other parameters. Recall that the diversity between the products was calculated for *Stopk* using all vectors in W and not only the ones used for the identification of the products.

Figures 5(e)–5(h) show the execution time of the algorithms with respect to various parameters. In Figure 5(e) we notice that none of the algorithms is influenced significantly by the data set cardinality, because top-k queries require retrieving only few data objects that are highly ranked independently of the data set cardinality. Figure 5(f) (which uses log-scale) shows that both algorithms are influenced in a similar way while varying the number of weighting vectors $|W|$, but *Stopk* is always more efficient than *Atopk*. In particular, the time increases with increasing number of weighting vectors $|W|$. In Figure 5(g), we depict the performance of both algorithms for different data distributions. In the case of anticorrelated data the execution time for *Atopk* is 4 times larger that in the case of a uniform distribution. On the other hand, the performance of *Stopk* is not affected by anticorrelated values significantly. Figure 5(h) shows that *Stopk* scales nicely for increased dimensionality, in contrast to *Atopk* whose cost increases by one order of magnitude when we go from 3 to 6 dimensions. This experiment provides strong evidence for the scalability of *Stopk* with increased number of dimensions.

Figures 5(i)–5(l) evaluate the performance of our algorithms in terms of I/O accesses. The conclusions for the I/O accesses are similar to those for the time, except for the case of increasing the data set cardinality $|S|$. In Figure 5(i), we notice that even though the time is not increased by varying $|S|$, the I/O accesses increase. Naturally, more I/O are needed for processing a top-k query of a larger data set, but this is not reflected on the time. Due to the buffering, the computational cost of processing multiple top-k queries is more significant than the time needed to retrieve the relevant index nodes. Nevertheless, in all cases *Stopk* outperforms *Atopk* in terms of I/O. For example Figure 5(l) shows that *Stopk* is two orders of magnitude cheaper than *Atopk* in I/O accesses as we increase the dimensions.



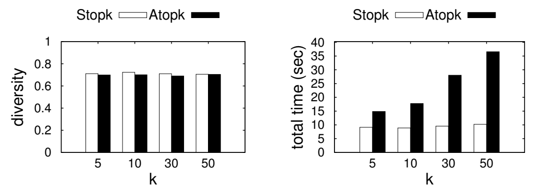
(a) Diversity



(b) Time

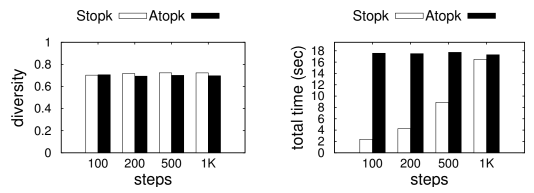
Figure 6: Varying r.

Varying r. Figure 6 examines the effect of varying the number r of retrieved products on our algorithms. First, we observe in Figure 6(a) a decreasing tendency of the diversity value as r increases for both algorithms, which is expected as also the diversity value of the optimal solution will decrease as the most diverse products are selected first. As far as the performance is considered, in Figure 6(b), the time of *Atopk* is not influenced by the increase of r, because *Atopk* computes all top-k queries independently of the size of the result set r and the computational cost of *DPSA* algorithm



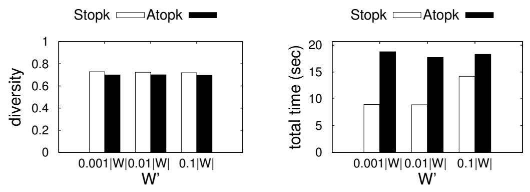
(a) Diversity (b) Time

Figure 7: Varying *k*.



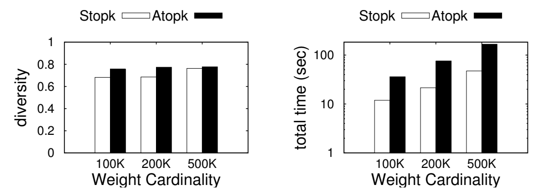
(a) Diversity (b) Time

Figure 8: Varying *steps*.



(a) Diversity (b) Time

Figure 9: Varying W'.



(a) Diversity: varying |W| (b) Time: varying $|W|$

Figure 10: House Dataset: varying |W|

is not significant compared to the cost of the top-k queries. *Stopk* is also not significantly affected, since the values of r are relatively small, and the algorithm is executed *steps* times in any case. Still, *Stopk* remains always much faster than *Atopk*.

Varying k. In Figure 7, we gradually increase the parameter k of the reverse top-k queries from 5 to 50. In Figure 7(a), we notice that the diversity value is stable as k increases, which seems counter-intuitive at first. By increasing k the size of the reverse top-k set increases for some objects and more objects have a non-empty reverse top-k set. However, this does not influence the diversity value significantly, as the most diverse centroids may not change. Figure 7(b) depicts the time obtained for different values of k. Although we witness a small deterioration in the performance of both algorithms, *Stopk* consistently outperforms *Atopk*. Processing top-k queries is more time-consuming for higher values of k and the *DPSA* algorithm gets slower with increasing k because the number of candidates for finding the diverse objects increases. We should add however, that the effect of parameter k has much smaller impact on the performance of *Stopk* because *Stopk* performs a small number of top-k queries.

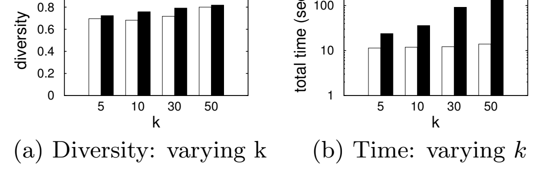
Varying steps. The *steps* parameter is an essential parameter for the *Stopk* algorithm as it balances the efficiency of the algorithm and the diversity the algorithm achieves. Recall that *Stopk* performs only *steps* top-k queries, which is only a small fraction of the $|W|$ top-k queries that *Atopk* performs. On the other hand, *Stopk* executes also *steps* times *DPSA* algorithm algorithm on a small set of approximate centroids, which is not necessary for *Atopk*. In Figure 8, we observe that the diversity achieved using very few vectors is quite close to the diversity achieved by *Atopk*. As we increase the *steps* parameter the achieved diversity increases marginally. However the execution time increases proportionally with the increase of the *steps* parameter. This experiment verifies that a small value of *steps* is sufficient to produce results of high diversity in a very efficient way.

Varying sample size $|W'|$. The size of sample of preferences from which we select the two initial centroids plays an important role in the performance of the *Stopk* algorithm. The complexity of the selection process is $O(|W'|^2)$ and therefore a large sample can have significant impact on the performance of the algorithm. However, as shown in Figure 9, the increased cost in performance is not accompanied by an increased gain in diversity. The reason behind this fact is that once the sample is large enough to offer a good representation of the whole set of preferences, further enlargement will not help significantly in finding better initial centroids.

9.4 Results on Real Data

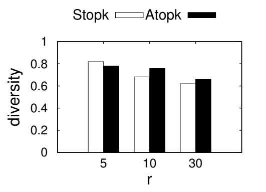
We have also performed an evaluation of our algorithm using a real data set. The conclusions drawn are overall in accordance with the conclusions made by the evaluation with synthetic data, thus verifying our findings. The default setup for this series of experiments is $|S| = 127930$, $|W| = 100K$, $k = 30$, $r = 10$, W:UN, $steps = 500$, $s = 0.01 \cdot |W|$. The size of the data set used and the high complexity of the exact algorithm (*opt*) did not allow the exact algorithm to terminate and therefore we did not include its performance results in this series of experiments.

Analysis for varying $|W|, k$, and r. Figures 10-12 depict performance of the two algorithms. For all values of the varying parameters *Stopk* achieves diversity values close to the ones of *Atopk*. For both algorithms we notice a drop in the diversity values when r is increases which is expected as analyzed in 9.1. With respect to processing time it is evident

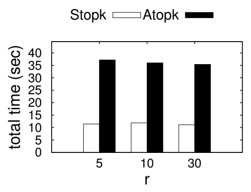


(a) Diversity: varying k (b) Time: varying k

Figure 11: House Dataset: varying *k*



(a) Diversity: varying r



(b) Time: varying r

Figure 12: House Dataset: varying r

that both parameters $|W|$ and k play a significant role in the performance of *Atopk*. This does not come as a surprise as the processing cost of *Atopk* is dominated by the processing cost of the top-k queries needed for the computation of the centroids of the $RTOP_k$ sets for each product. On the contrary *Stopk* is much less affected by those parameters as it performs a limited number of top-k queries. The increase of parameter r has little effect in both algorithms. The performance difference with respect to I/O is in all cases larger than two orders of magnitude. Only exception is for $k < 10$ where *Stopk* is one order of magnitude more efficient.

10. CONCLUSIONS

In this paper, we address the important problem of selecting the r most diverse products based on customers' preferences. The reverse top-k set of each product is represented by its centroid and the distance between centroids is then expressed using cosine distance. In order to find products that are attractive to customers with dissimilar preferences, we define the r-Diversity problem as a *dispersion* problem applied on the products' reverse top-k sets. As dispersion problems are known to be NP-hard, we propose two approximate algorithms that solve the problem. The first algorithm computes the reverse top-k sets and then applies a greedy algorithm that retrieves a set of products of high diversity. The second applies the greedy algorithm on an approximation of the reverse top-k sets by evaluating only some carefully selected top-k queries. In our experimental evaluation, we study the performance of the proposed algorithms and the diversity of the retrieved products in various experimental setups. In particular, we demonstrate that our algorithms both achieve diversity values close to optimal and are very efficient in practice.

Acknowledgments

The work of A. Vlachou was supported by the Action "Supporting Postdoctoral Researchers" of the Operational Program "Education and Lifelong Learning" (Action's Beneficiary: General Secretariat for Research and Technology), and is co-financed by the European Social Fund (ESF) and the Greek State. The research of C. Doulkeridis has been co-financed by ESF and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: Aristeia II, Project: ROADRUNNER.

11. REFERENCES

[1] A. Angel and N. Koudas. Efficient diversity-aware search. In *Proc. of SIGMOD*, pages 781–792, 2011.
[2] A. Arvanitis, A. Deligiannakis, and Y. Vassiliou. Efficient influence-based processing of market research queries. In *Proc. of CIKM*, pages 1193–1202, 2012.
[3] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *Proc. of ICDE*, pages 421–430, 2001.
[4] S. Chester, A. Thomo, S. Venkatesh, and S. Whitesides. Indexing reverse top-k queries in two dimensions. In *Proc. of DASFAA (1)*, pages 201–208, 2013.
[5] M. Drosou and E. Pitoura. Diversity over continuous data. *IEEE Data Eng. Bull.*, 32(4):49–56, 2009.
[6] M. Drosou and E. Pitoura. DisC diversity: result diversification based on dissimilarity and coverage. *PVLDB*, 6(1):13–24, 2012.
[7] M. Drosou and E. Pitoura. Dynamic diversification of continuous data. In *Proc. of EDBT*, pages 216–227, 2012.
[8] E. Erkut. The discrete p-dispersion problem. *European Journal of Operational Research*, 46(1):48–60, 1990.
[9] S. Ge, L. H. U, N. Mamoulis, and D. W. Cheung. Efficient all top-k computation - a unified solution for all top-k, reverse top-k and top-m influential queries. *TKDE*, 25(5):1015–1027, 2013.
[10] J.-L. Koh, C.-Y. Lin, and A. Chen. Finding k most favorite products based on reverse top-t queries. *The VLDB Journal*, pages 1–24, 2013.
[11] C. Li, B. C. Ooi, A. K. H. Tung, and S. Wang. DADA: a data cube for dominant relationship analysis. In *Proc. of SIGMOD*, pages 659–670, 2006.
[12] C.-Y. Lin, J.-L. Koh, and A. L. P. Chen. Determining (k)-most demanding products with maximum expected number of total customers. *IEEE Trans. Knowl. Data Eng.*, 25(8):1732–1747, 2013.
[13] X. Lin, Y. Yuan, Q. Zhang, and Y. Zhang. Selecting stars: The k most representative skyline operator. In *Proc. of ICDE*, pages 86–95, 2007.
[14] M. Miah, G. Das, V. Hristidis, and H. Mannila. Standing out in a crowd: Selecting attributes for maximum visibility. In *Proc. of ICDE*, pages 356–365, 2008.
[15] S. S. Ravi, D. J. Rosenkrantz, and G. K. Tayi. Heuristic and special case algorithms for dispersion problems. *Operations Research*, 42(2):299–310, 1994.
[16] Y. Tao, L. Ding, X. Lin, and J. Pei. Distance-based representative skyline. In *Proc. of ICDE*, pages 892–903, 2009.
[17] Y. Tao, V. Hristidis, D. Papadias, and Y. Papakonstantinou. Branch-and-bound processing of ranked queries. *Inf. Syst.*, 32(3):424–445, 2007.
[18] G. Valkanas, A. N. Papadopoulos, and D. Gunopulos. SkyDiver: a framework for skyline diversification. In *Proc. of EDBT*, pages 406–417, 2013.
[19] M. R. Vieira, H. L. Razente, M. C. N. Barioni, M. Hadjieleftheriou, D. Srivastava, C. T. Jr., and V. J. Tsotras. DivDB: a system for diversifying query results. *PVLDB*, 4(12):1395–1398, 2011.
[20] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Nørvåg. Reverse top-k queries. In *Proc. of ICDE*, pages 365–376, 2010.
[21] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Nørvåg. Monochromatic and bichromatic reverse top-k queries. *TKDE*, 23(8):1215–1229, 2011.
[22] A. Vlachou, C. Doulkeridis, K. Nørvåg, and Y. Kotidis. Identifying the most influential data objects with reverse top-k queries. *PVLDB*, 3(1):364–372, 2010.
[23] A. Vlachou, C. Doulkeridis, K. Nørvåg, and Y. Kotidis. Branch-and-bound algorithm for reverse top-k queries. In *Proc. of SIGMOD*, pages 481–492, 2013.
[24] Q. Wan, R. C.-W. Wong, I. F. Ilyas, M. T. Özsu, and Y. Peng. Creating competitive products. *PVLDB*, 2(1):898–909, 2009.
[25] T. Wu, Y. Sun, C. Li, and J. Han. Region-based online promotion analysis. In *Proc. of EDBT*, pages 63–74, 2010.
[26] T. Wu, D. Xin, Q. Mei, and J. Han. Promotion analysis in multi-dimensional space. *PVLDB*, 2(1):109–120, 2009.
[27] A. Yu, P. K. Agarwal, and J. Yang. Processing a large number of continuous preference top-k queries. In *Proc. of SIGMOD*, pages 397–408, 2012.

Index Design for Enforcing Partial Referential Integrity Efficiently*

Mozhgan Memari
Department of Computer Science
University of Auckland, New Zealand
m.memari@auckland.ac.nz

Sebastian Link
Department of Computer Science
University of Auckland, New Zealand
s.link@auckland.ac.nz

ABSTRACT

Referential integrity is fundamental for data processing and data quality. The SQL standard proposes different semantics under which referential integrity can be enforced in practice. Under simple semantics, only total foreign key values must be matched by some referenced key values. Under partial semantics, total and partial foreign key values must be matched by some referenced key values. Support for simple semantics is extensive and widespread across different database management systems but, surprisingly, partial semantics does not enjoy any native support in any known systems. Previous research has left open the questions whether partial referential integrity is useful for any real-world applications and whether it can enjoy efficient support at the systems level. As our first contribution we show that efficient support for partial referential integrity can provide database users with intelligent query and update services. Indeed, we regard partial semantics as an effective imputation technique for missing data in query answers and update operations, which increases the quality of these services. As our second contribution we show how partial referential integrity can be enforced efficiently for real-world foreign keys. For that purpose we propose triggers and exploit different index structures. Our experiments with synthetic and benchmark data sets confirm that our index structures do not only boost the performance of the state-of-the-art recommendation for enforcing partial semantics in real-world foreign keys, but show trends that are similar to enforcing simple semantics.

1. INTRODUCTION

In his seminal paper [5] Codd introduced the principles of entity and referential integrity as two fundamental cornerstones of the relational model of data. While more than 100 classes of relational integrity constraints have been investigated [21], relational database management systems offer only native support for keys and foreign keys, which enforce entity and referential integrity, respectively. Indeed, keys and foreign keys provide principled mechanisms to process quality data efficiently. The SQL standard promotes the use of two different semantics for referential integrity. Under *simple* semantics, referencing tuples with null markers on some of their foreign key columns satisfy referential integrity by default. Under *partial* semantics, every referencing tuple requires a referenced tuple that matches all total values on the foreign key columns in the corresponding key columns. Partial semantics offers a higher degree of data quality as it subsumes simple semantics.

EXAMPLE 1. *For illustration consider an example from an Australian tourism company [8]. Tours in the* TOUR *table have a tour_id, for example a tour such as the "Gold Coast Grand Tour" has tour_id GCG. Tours have a fixed sequence of sites they visit. Sites are identified by a site_code (e.g., MV) but also have a unique site_name (e.g., Movie World). The primary key on* TOUR *is* {*tour_id, site_code*}. *Booking orders by visitors, who can join a tour from any allocated site, are stored in the* BOOKING *table. The foreign key*

$$[tour\_id, site\_code] \subseteq \text{TOUR}[tour\_id, site\_code]$$

is defined on BOOKING. *Consider the following database.*

TOUR

Tour_id	***Site_code***	***Site_name***
GCG	*OR*	*O'Reilly's*
BRT	*OR*	*O'Reilly's*
BRT	*MV*	*Movie World*
RF	*BB*	*Binna Burra*
RF	*OR*	*O'Reilly's*

BOOKING

Visitor_id	***Tour_id***	***Site_code***	***Date***
1006	*BRF*	`null`	*Sep* 19^{th}
1001	*BRT*	*OR*	*Nov* 21^{st}
1008	`null`	*BB*	*Sep* 5^{th}
1012	`null`	*BR*	*Nov* 2^{nd}
1011	*RF*	`null`	*Oct* 5^{th}

Simple referential integrity is satisfied: the only total foreign key value (BRT,OR) in the BOOKING *table is matched in the* TOUR *table. Partial referential integrity is violated: for the partial foreign key value (BRF,*`null`*) in the* BOOKING *table there is no tuple in the* TOUR *table with value "BRF" on tour_id, and similarly for (*`null`*, BR).*

While every database management system we know offers built-in support to enforce simple referential integrity,

*This research is supported by the Marsden fund council from Government funding, administered by the Royal Society of New Zealand.

10.5441/002/edbt.2015.20

it is surprising that none of them offers built-in support to enforce partial referential integrity [22]. Explanations for this gap between the SQL standard and its implementations have been brought forward by Härder and Reinhart, who analyzed in great detail the requirements of partial referential integrity on the operational level [9]. Two main questions remain open two decades after they have been posed in [9]:

1. Is partial referential integrity useful for any real-world application?
2. Can partial referential integrity be enforced efficiently?

Contribution. In the present paper we provide affirmative answers to both questions. Our first main contribution shows that partial referential integrity is useful for the two most significant real-world applications of database technology: updates and queries. More specifically, we propose intelligent update and query services that are based on efficiently enforcing partial semantics. Indeed, partial referential integrity can be exploited to impute missing data values. Our intelligent updates provide database users with sound choices to reduce the level of incompleteness in the database, which is an important measure for the quality of data [7]. In the example above, suppose that the last tuple is not already part of the BOOKING table but about to be inserted. Based on the assumption that the updated table shall satisfy partial referential integrity, our update service would provide the user with the option to replace the null marker by either "BB" or "OR". This service can be customized further depending on the authorization rights of the user, for example. Our intelligent query service provides database users with additional query answers that result from the imputation of missing data values in standard answers. In our example, the standard answer to the query that selects *tour_id* and *site_code* from tuples in the BOOKING table, may be augmented by the tuples (RF,BB), and (RF,OR) based on the partial semantics of the foreign key. We believe that intelligent updates and queries offer a strong affirmative answer to the first open question, that goes beyond the straightforward argument that partial referential integrity targets a higher level of data quality than simple referential integrity. Note that our applications cannot be supported by simple semantics. Indeed, they provide strong motivation to investigate the second open question, as the effectiveness of the applications largely depend on the efficiency of enforcing partial semantics for real-world foreign keys. Based on our experience, the literature [6] and public schemata, most real-world foreign keys have rarely more than four columns and the referenced key is commonly the primary key, or a candidate key where all columns are `NOT NULL`. Our second main contribution is a detailed analysis of partial referential integrity at the systems level, as proposed for future work in [9]. The main finding is that partial semantics for "real-world" foreign keys can be implemented in the form of triggers and enforced efficiently by a right combination of indices. In general, it is worthwhile investigating which subsets of the foreign key columns carry the most total values, and then define indices on those subsets. As updates include the maintenance of all affected indices, having too many indices means that the loss in time for their maintenance outweighs the gain in search time when enforcing referential integrity. Some organizations may have a good knowledge of the top-k indices they want to support, but we have made it part of our research to shed further light on what a reasonable number k could be. For this purpose, we applied our analysis to test data in which null marker occurrences are evenly distributed between all possible subsets of columns. That is, we have the least degree of information available about which indices to define. Our recommendation for an n-column foreign key is to exploit $n + 1$ indices on each of the referencing and referenced tables: one compound index on the n columns, and one index on each of the n individual columns. This combination of indices outperforms any other combination for all possible kinds of updates. The finding is confirmed by experiments on two benchmark and one real life database. We remark that the original proposal by Härder and Reinhart to utilize one index for each of the n key columns on the referenced table and one compound index on the foreign key columns of the referencing table performs well only for 2-column key relationships. Essentially, by doubling the number of indices over their proposal, we improve the speed for inserts by a factor of 7, and for deletions by a factor of 123, on the largest data set considered with a 5-column foreign key relationship. Note that this includes the maintenance of all indices involved. Further experiments explain this performance boost over the original proposal: Adding an index for each of the n foreign key columns on the referencing table overcomes the poor performance of the original proposal when deleting tuples from the referenced table that have no alternative match for referencing tuples. Furthermore, adding a single index for the compound key on the referenced table boosts the performance when inserting total tuples in the referencing table. The only trade-off we found is that the time for loading data on the referencing table is 1.5 times more, due to building twice as many indices. This one-time cost is feasible.

Organization. The remainder of this paper is organized as follows. We comment on some related work in Section 2. Details on the semantics of referential integrity constraints in SQL are given in Section 3. We describe our ideas of intelligent update and query services in Sections 4 and 5, based on partial semantics. In Section 6 we propose triggers as well as five different index structures for which we will analyze the performance of enforcing partial referential integrity on synthetic data sets in Section 7, and on TPC-C and TPC-H data sets in Section 8. We conclude in Section 9 where we also briefly comment on future work.

2. RELATED WORK

Work on referential integrity has largely addressed inclusion dependencies. A seminal paper on the theory of inclusion dependencies is [2], in which a finite axiomatization for the associated implication problem is presented and the non-k-ary-axiomatizability of both finite and unrestricted implication for functional and inclusion dependencies together is demonstrated. An axiomatization which is not k-ary for the finite implication of functional and inclusion dependencies is presented in [18]. Undecidability of (finite) implication for functional and inclusion dependencies taken together was shown independently by [4] and [19]. Another seminal paper is [12], which also observed the distinction between finite and unrestricted implication for functional and inclusion dependencies, generalized the chase to incorporate functional and inclusion dependencies, and used this to characterize containment between conjunctive queries. Databases have benefited from referential integrity constraints and inclusion dependencies in areas as diverse as database design [15], consistency enforcement [3], query optimization [12], data cleaning [1], data quality [20], and data profiling [24]. Inclusion dependencies have also been considered in XML [13] and RDF [14].

The different semantics of referential integrity, as proposed by the SQL standard [17], have not received much attention from neither academia nor practice. As observed in [22], there are no database management systems that offer built-in support for enforcing partial referential integrity while every database management system offers built-in support for enforcing simple referential integrity. In [9] Härder and Reinhart investigated the functional requirements for preserving simple and partial referential integrity. Indeed, they determined the number and kinds of searches necessary for referential integrity maintenance, without implementation considerations. Their main result was that a combined access path structure is the most appropriate for checking simple semantics, while partial semantics requires very expensive and complicated check procedures. Their "best advice is to avoid the use of `MATCH PARTIAL` at all". If required, they recommend the use of one index for each of the key columns on the referenced table and one compound index on the foreign key columns of the referencing table. They also investigate the performance of multi-dimensional access paths by considering grid file structures. Here, the access costs for partial match queries are remarkably more expensive than their suggested index option. The main reason is that grid files retrieve all matching tuples while partial referential integrity requires only one matching tuple. In conclusion, Härder and Reinhart say that their "presented results rely on the assumption that the search costs are indicative for the entire costs of referential integrity maintenance. This assumption has to be justified through further research especially at the system level. Another interesting question to be answered is whether or not `MATCH PARTIAL` is useful for a real world application". Here, we address both questions.

In a research-in-progress paper [16] we performed a static analysis of the costs for validating simple and partial semantics in a fixed database. Therefore, the analysis did not consider updates at all. It only applied to referential integrity constraints with two columns, and only considered compound indices which refer to all columns of a foreign key. The present paper presents a detailed analysis of the costs for enforcing simple and partial semantics in a dynamic database that is subject to updates; applies to foreign keys with up to five columns, considers multiple index structures on referenced and referencing tables, and proposes triggers and the new intelligent query and update services.

3. REFERENTIAL INTEGRITY IN SQL

Foreign keys form one of the most fundamental classes of integrity constraints, which implement Codd's proposal of referential integrity from his seminal paper [5]. Referential integrity maintains the relationship between two table schemata, which are the referencing schema or child schema, usually denoted by C_S, and the referenced schema or parent schema, usually denoted by P_S. A referential integrity constraint is commonly written as

$$[f_1, \ldots, f_n] \subseteq P_S[k_1, \ldots, k_n]$$

to denote the relationship between the sequence $[f_1, f_2, \ldots, f_n]$ of distinct column names on C_S, usually called the foreign key, and the sequence $[k_1, k_2, \ldots, k_n]$ of distinct column names, which form a candidate key on P_S. For $i = 1, \ldots, n$, the domains of the column names f_i and k_i must match. Intuitively, referential integrity requires that for each tuple c in a child table C there is a matching tuple p in the parent table P. The SQL standard recommends the use of the `MATCH` clause to specify different ways for handling occurrences of the null marker `null` in foreign key and key columns [17].

Under *simple* semantics, the foreign key constraint is satisfied if for every tuple c in the child table C, either $c(f_i) = \texttt{null}$ for some $1 \leq i \leq n$, or there is some tuple p in the parent table P such that $c(f_i) = p(k_i)$ for all $i = 1, \ldots, n$. More precisely,

$$\forall c \in C \left(\left(\bigwedge_{i=1}^{n} c(f_i) \neq \texttt{null} \right) \Rightarrow \exists p \in P \left(\bigwedge_{i=1}^{n} c(f_i) = p(k_i) \right) \right) .$$

Hence, simple referential integrity is never violated by tuples that are partially defined on the foreign key columns.

Under *partial* semantics, the foreign key constraint is satisfied if for every tuple c in the child table C there is some tuple p in the parent table P such that $c[f_1, \ldots, f_n]$ is subsumed by $p[k_1, \ldots, k_n]$. That is,

$$\forall c \in C \exists p \in P\, (c[f_1, \ldots, f_n] \sqsubseteq p[k_1, \ldots, k_n])$$

Here, a tuple c over the column sequence $[f_1, \ldots, f_n]$ is *subsumed* by a tuple p over the column sequence $[k_1, \ldots, k_n]$, if for every $i = 1, \ldots, n$, $c(f_i) = \texttt{null}$ or $c(f_i) = p(k_i)$.

The table from Example 1 satisfies the foreign key constraint [*tour_id*,*site_code*] ⊆TOUR [*tour_id*,*site_code*] on table BOOKING under simple semantics, but not under partial semantics. For instance, the BOOKING-tuple (BRF,`null`) over [*tour_id*,*site_code*] has no TOUR-tuple over [*tour_id*, *site_code*] by which it is subsumed.

Enforcing some referential integrity constraint means that each time the child or parent table is modified it must be verified that the constraint is satisfied by the modified database instance. Therefore, referential integrity is particularly important for transactional databases, or for updates of important data such as master data.

In general, six basic updates must be addressed to accommodate all possible modifications on parent tables P or child tables C. Tuple inserts into P and tuple deletions from C do not cause a violation of referential integrity. The other four update operations, however, ⟨*Insert a new tuple into C*⟩ and ⟨*Update C*⟩, ⟨*Update P*⟩ and ⟨*Delete a tuple from P*⟩ may lead to modified tables that violate referential integrity. If a tuple c from C references a tuple p from P, we call c a child of p, and p a parent of c.

⟨*Delete a tuple from P*⟩: A tuple p from P may be the only parent of some child c from C. That is, there is no other tuple p' in P which is a parent of c. Deleting such single parents from P will always violate referential integrity.

⟨*Update P*⟩: This update can be interpreted as ⟨*Delete a tuple from P*⟩ along with ⟨*Insert a new tuple into P*⟩. Therefore, it may only cause a referential integrity violation due to the Delete action.

⟨*Insert a new tuple into C*⟩: Referential integrity requires for every newly inserted child c in C the existence of a parent p in P. Otherwise, referential integrity is violated.

⟨*Update C*⟩: An update of a child in C can be interpreted as a delete from C followed by an insert into C. Only the insert into C can lead to a violation of referential integrity.

According to the SQL standard, inserts into C or updates on C are only allowed if they result in a new child table that satisfies the referential integrity constraints defined on C_S. However, different actions can be applied when a delete from P or an update on P results in the violation of some referential integrity constraint. Based on the SQL standard, "CASCADE", "SET NULL", "SET DEFAULT", "RESTRICT" and "NO ACTION" are available referential actions.

Under simple semantics, every child has at most one parent. Under partial semantics, any child may have several parents, given that the child has a null marker occurrence in some of its foreign key columns. If the state of a child is defined as the subset of the n foreign key columns on which it is `null`, then each given parent may have up to $2^n - 1$ children that have pairwise different states. If u denotes the number of null marker occurrences ($0 \leq u < n$), $\binom{n}{u}$ is the number of distinct states with u null marker occurrences [9].

Example 2. *Given a 3-attribute key with value $\langle 1,2,3\rangle$ on the key columns, the seven different states may result from children with the following values on their foreign key columns: $\langle 1,2,3\rangle$, $\langle null,2,3\rangle$, $\langle 1,null,3\rangle$, $\langle 1,2,null\rangle$, $\langle null, null,3\rangle$, $\langle null,2,null\rangle$ and $\langle 1,null,null\rangle$. Each of the non-total children may have some other parent. The parent with $\langle 4,2,3\rangle$, for example, has also children with the following values on their foreign key columns $\langle null,2,3\rangle$, $\langle null,2,null\rangle$ and $\langle null,null,3\rangle$.*

4. AN INTELLIGENT UPDATE SERVICE

We propose an intelligent update service that enables us to give an affirmative answer to Härder and Reinhart's question "whether or not `MATCH PARTIAL` is useful for a real world application?" [9].

Null markers offer great flexibility for data entry, but have serious consequences. Indeed, the level of information completeness is an important factor for data quality [7]. Partial data causes significant problems for enterprises: it routinely leads to misleading analytical results and biased decisions, and accounts for loss of revenue, credibility and customers [7]. We propose the use of `MATCH PARTIAL` for intelligent updates of data. For a given foreign key $[f_1,\ldots,f_n] \subseteq P_S[k_1,\ldots,k_n]$ on C_S we propose the following two strategies to reduce information incompleteness.

4.1 Intelligent Insertions

Suppose a new tuple c is inserted into the child table C and $c(f_i) =$ `null` on some foreign key column $f_1,\ldots,f_n$. Then the database management system (DBMS) determines all parents p of the parent table P where $c[f_1,\ldots,f_n] \sqsubseteq p[k_1,\ldots,k_n]$. For each of these parents p, the DBMS computes the tuple c_p that results from c by replacing each null marker occurrence $c(f_i) =$ `null` by the value $p(k_i)$, where $i = 1,\ldots,n$. Finally, the tuples c_p are presented as alternatives to c for insertion into C.

For instance, suppose again that the tuple $c =$ (1011, RF, `null`, Oct 5th) is not already part of the Booking table in Example 1, but about to be inserted into it. Our intelligent update service would determine all possible parents of c in Tour, which are $p =$ (RF, BB, Binna Burra) and $p' =$ (RF, OR, O'Reilly's), and present the tuples $c_p =$ (1011, RF, BB, Oct 5th) and $c_{p'} =$ (1011, RF, OR, Oct 5th) as alternatives to $c =$ (1011, RF, `null`, Oct 5th) for insertion into Booking.

4.2 Intelligent Deletions

Suppose an existing tuple p is deleted from P. For all children c of p in C where $c[f_1,\ldots,f_n] \sqsubseteq p[k_1,\ldots,k_n]$ and $c(f_i) =$ `null` on some $f_1,\ldots,f_n$, the DBMS determines all alternative parents of p in $P - \{p\}$, i.e. those tuples $p' \in P - \{p\}$ where $c[f_1,\ldots,f_n] \sqsubseteq p'[k_1,\ldots,k_n]$. It then computes the tuple $c_{p'}$ that results from c by replacing each null marker occurrence $c(f_i) =$ `null` by the value $p'(k_i)$, where $i = 1,\ldots,n$. Finally, for each child c of p the tuples $c_{p'}$ are presented as potential updates of c in C.

For instance, consider the tables from Example 1. Suppose the tuple $p =$ (RF, OR, O'Reilly's) is deleted from the Tour table. The only child of p in Booking is $c =$ (1011, RF, `null`, Oct 5th), and the only alternative parent of c in Tour is $p' =$ (RF, BB, Binna Burra). Consequently, the DBMS presents the tuple $c_{p'} =$ (1011, RF, BB, Oct 5th) as an update of the tuple c in Booking.

We propose two different approaches for implementing intelligent deletions. In Method 1, the existence of alternative parents is first checked in P. Then the potential updates are ranked according to the number of affected children in C and presented to the user for confirmation.

Algorithm 1 Intelligent Deletion: Method 1

```
Require: Deleted tuple: $p[k_1,\ldots,k_n]$, referential action
Ensure: Updated $C$ under Partial Semantics
 1: forall $c \in \mathbf{C}$ such that $\bigwedge_{i=1}^{n} c(f_i) \neq$ null and
    $\bigwedge_{i=1}^{n} c(f_i) = p(k_i)$ do Apply referential action
 2: for u:=1 to n-1 do
 3:     $S[u] \leftarrow \{S_{uj} |$ the $j^{th}$ state of $\langle k_1,\ldots,k_n\rangle$ with $u$
        nulls, for all j=1 to $\binom{n}{u}\}$
 4:     for all $S_{uj} \in S[u]$ do
 5:         $Q[S_{uj}] \leftarrow \{p' \in P - \{p\} | S_{uj} \sqsubseteq p'[k_1,\ldots,k_n]\}$
 6:         $l_{uj} \leftarrow$ number of $c$ in $C$ match $S_{uj}$
 7:         $l_{mj} \leftarrow$ number of $c$ in $C$ match $S_{mj}$ such that
            $m = u+1$ to $n-1$ and $S_{mj} \sqsubseteq S_{uj}$
 8:         $l'_{uj} \leftarrow l_{uj} + l_{mj}$
 9:         if $Q[S_{uj}] = \emptyset$ and $l_{uj} \neq 0$ then
10:             Apply referential action on $S_{uj}$ states in $C$
11:             $l'_{uj} \leftarrow 0$
12:     $L \leftarrow \{(l'_{uj}, Q[S_{uj}]\}$
13:     if $\exists\, l'_{uj} \neq 0 \in L$ then
14:         $S'_u \leftarrow S_{uj}$ with Max($l'_{uj}$)
15:         Output: Set $Q[S'_u]$ and $S'_u$ (foreign key)
16:         Input: If $p' \in Q[S'_u]$ is selected then Subsume all
            $c = S_{uj}$ and $c = S_{mj}$ by $p'$
17:         $l'_{uj} \leftarrow 0$ and $L \leftarrow \{(l'_{uj}, Q[S'_u]\}$
```

In Method 2, the intelligent system first finds all children of the given parent. For each of these children, the system then finds all alternatives. The choice of method depends on the requirements of the application.

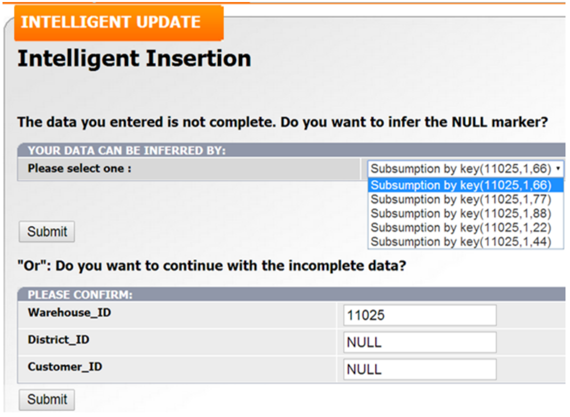
4.3 Implementation

The system is available on `http://sqlkeys.info` and has been tested on the 3-column foreign key of the TPC-C benchmark database from the Transaction Processing Performance Council (`http://www.tpc.org`). Figure 1 shows how users can insert either their original record or a more informative record, whatever is perceived to be the better choice. Figures 2 and 3 show screenshots of the two deletion methods. Indeed, the choice of continuing with incomplete foreign keys

Algorithm 2 Intelligent Deletion: Method 2

```
Require: Deleted tuple: $p[k_1, \ldots, k_n]$, referential action
Ensure: Updated $C$ under Partial Semantics
 1: forall $c \in \mathbf{C}$ such that $\bigwedge_{i=1}^{n} c(f_i) \neq$ null and
    $\bigwedge_{i=1}^{n} c(f_i) = p(k_i)$ do Apply referential action
 2: for u:=1 to n-1 do
 3:     $S[u] \leftarrow \{S_{uj}|$ the $j^{th}$ state of $\langle k_1, \ldots, k_n \rangle$ with $u$
        nulls, for all j=1 to $\binom{n}{u}\}$
 4:     for all $S_{uj} \in S[u]$ do
 5:         $l_{uj} \leftarrow$ number of $c$ in $C$ match $S_{uj}$
 6:     $L \leftarrow \{(l_{uj}, Q[S_{uj}]\}$
 7:     if $\exists\, l_{uj} \neq 0 \in L$ then
 8:         $S'_u \leftarrow S_{uj}$ with Max$(l_{uj})$
 9:         $Q[S'_u] \leftarrow \{p' \in P - \{p\} | S'_u \sqsubseteq p'[k_1, \ldots, k_n]\}$
10:         if $Q[S'_u] = \emptyset$ then
11:             Apply referential action on $S'_u$ states in $C$
12:         else
13:             Output: Set $Q[S'_u]$ and $S'_u$ (foreign key)
14:             Input: If $p' \in Q[S'_u]$ is selected then Subsume
                all $c = S_{uj}$ and $c = S_{mj}$ by $p'$ where $m = u+1$
                to $n-1$ and $S_{mj} \sqsubseteq S_{uj}$
15:             $l_{uj} \leftarrow 0$ and $L \leftarrow \{(l_{uj}, Q[S'_u]\}$
```

Figure 1: Intelligent Update System: Insertion


INTELLIGENT UPDATE

Intelligent Insertion

The data you entered is not complete. Do you want to infer the NULL marker?

YOUR DATA CAN BE INFERRED BY:

Please select one :	Subsumption by key(11025,1,66)
	Subsumption by key(11025,1,66)
	Subsumption by key(11025,1,77)
	Subsumption by key(11025,1,88)
	Subsumption by key(11025,1,22)
	Subsumption by key(11025,1,44)

Submit

"Or": Do you want to continue with the incomplete data?

PLEASE CONFIRM:

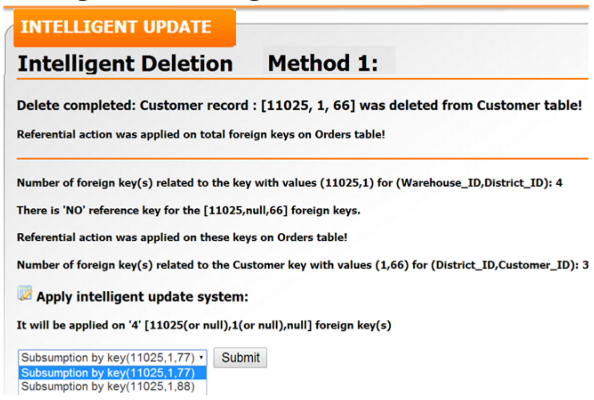
Warehouse_ID	11025
District_ID	NULL
Customer_ID	NULL

Submit

is available to users, however, referential action will be applied on the foreign keys which violate partial referential integrity.

Use cases. It is straightforward to envision novel use cases of the intelligent update service. For example, when updates are run manually, the user can be presented directly with available choices for the imputation of null markers. This can be customized further, for example, according to the preferred number of such choices or to the access rights the user enjoys. The decision should be based on the efficiency and quality requirements for data entry as well as the expertise of the user. When updates are run mechanically, it is particularly advisable to record the available choices for imputation in the form of a log. This log can be inspected later on for analytical purposes, or to assist with data cleaning. An interesting use case occurs whenever transactions are aborted due to null marker occurrences in child columns that are part of the primary key. Exploiting partial semantics to impute these occurrences by some matching consistent values may result in the successful completion of the transaction. In any case, the intelligent update service can help reduce information incompleteness in ways current services cannot.

Figure 2: Intelligent Deletion Method 1


INTELLIGENT UPDATE

Intelligent Deletion Method 1:

Delete completed: Customer record : [11025, 1, 66] was deleted from Customer table!

Referential action was applied on total foreign keys on Orders table!

Number of foreign key(s) related to the key with values (11025,1) for (Warehouse_ID,District_ID): 4

There is 'NO' reference key for the [11025,null,66] foreign keys.

Referential action was applied on these keys on Orders table!

Number of foreign key(s) related to the Customer key with values (1,66) for (District_ID,Customer_ID): 3

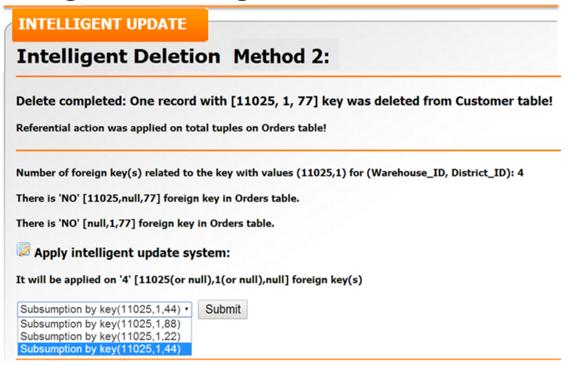
Apply intelligent update system:

It will be applied on '4' [11025(or null),1(or null),null] foreign key(s)

Subsumption by key(11025,1,77)
Subsumption by key(11025,1,77)
Subsumption by key(11025,1,88)

Submit

Figure 3: Intelligent Deletion Method 2


INTELLIGENT UPDATE

Intelligent Deletion Method 2:

Delete completed: One record with [11025, 1, 77] key was deleted from Customer table!

Referential action was applied on total tuples on Orders table!

Number of foreign key(s) related to the key with values (11025,1) for (Warehouse_ID, District_ID): 4

There is 'NO' [11025,null,77] foreign key in Orders table.

There is 'NO' [null,1,77] foreign key in Orders table.

Apply intelligent update system:

It will be applied on '4' [11025(or null),1(or null),null] foreign key(s)

Subsumption by key(11025,1,44)
Subsumption by key(11025,1,88)
Subsumption by key(11025,1,22)
Subsumption by key(11025,1,44)

Submit

5. AN INTELLIGENT QUERY SERVICE

The occurrence of null markers in query answers restricts the insights that stakeholders can gain from data. It is therefore important that database systems raise user awareness of actual data values that null markers may represent. By example, we will now explain why partial referential integrity constitutes a prime mechanism to reduce information incompleteness in query answering. Consider the following query which returns the *tour_id* and *site_code* of all bookings:

```
SELECT Tour_id, Site_code
FROM BOOKING
```

On our database from Example 1, the standard answer to this query consists of the records that are written in normal font below:

Tour_id	Site_code
BRF	null
BRT	OR
null	BB
RF	**BB**

Tour_id	Site_code
null	BR
RF	null
RF	**BB**
RF	**OR**

Partial semantics suggests to add the records written in bold font, as these have corresponding parents in the TOUR table. As illustrated above, users benefit from highlighting non-standard answers and placing them directly below the records in the standard answer from which they originate.

Summary. Our intelligent query and update services exploit partial semantics to minimize information incompleteness. This results in higher quality data, better data-driven decision making, and more competitive organizations. Both services complement each other: Fewer choices for imputation mean fewer choices for intelligent updates, and more choices for imputation mean more non-standard query answers users can benefit from. So, whichever case we encounter at least one of the services is useful. It is beyond the scope of this paper to go deeper into the specific application of our proposed services. Instead, we see them as strong drivers to investigate the second open question, that is, whether partial semantics can be enforced efficiently.

6. OPERATIONAL REQUIREMENTS

This section examines the two main operational requirements for enforcing partial semantics. First, we propose implementation details in the form of triggers on child and parent schemas. Next we discuss five different index structures. Their impact on the performance of enforcing partial referential integrity will be analyzed in subsequent sections.

6.1 Triggers for Partial Referential Integrity

Foreign keys commonly enforce simple semantics in current database management system implementations. Our next goal is to define triggers that enforce partial referential integrity under updates. For that purpose we first propose a trigger on the referencing child schema C_S. This trigger will enforce partial referential integrity for ⟨*Insert*⟩ and ⟨*Update*⟩ modifications on any child table C. The referential action we uniformly consider in our experiments is ⟨*SET NULL*⟩.

We have designed a platform on `www.sqlkeys.info` which generates triggers for enforcing partial semantics on any arbitrary database with foreign keys up to size five. Below is the SQL code for a trigger that implements a referential integrity constraint on $n = 3$ columns:

```
Trigger on C_S:
BEFORE INSERT ON C_S FOR EACH ROW
Declare msg varchar(80);
If (new.f1 is not null and new.f2 is not null and
    new.f3 is not null) then
  if not exists (select * from P_S where (k1=new.f1
  and k2=new.f2 and k3=new.f3)) then
  set msg ='No reference is found, enter a valid value';
  signal sqlstate '02000' set message_text = msg;
  end if;
Elseif (new.f1 is not null and new.f2 is not null and
  new.f3 is null) then if not exists(select * from P_S
  where (k1=new.f1 and k2=new.f2) LIMIT 1) then
  set msg ='No reference is found, enter a valid value';
  signal sqlstate '02000' set message_text = msg;
  end if;
Elseif (new.f1 is not null and new.f3 is not null and
  new.f2 is null) then if not exists (select * from P_S
  where (k1=new.f1 and k3=new.f3) LIMIT 1) then
  set msg ='No reference is found, enter a valid value';
  signal sqlstate '02000' set message_text = msg;
  end if;
Elseif ... /* similar for all 2^n − 1 possible states */
End if;
End;
```

For ⟨*Delete*⟩ and ⟨*Update*⟩ modifications on the parent schema P_S another trigger is defined. The SQL code of the trigger in the case of $n = 3$ columns is as follows:

```
Trigger on P_S:
AFTER DELETE ON P_S FOR EACH ROW
Update C_S set f1 =null, f2 =null, f3 =null where
    (old.k1 = f1 and old.k2 = f2 and old.k3 = f3);
If   exists (select * from C_S where (f2 is null and f3
   is null and old.k1 = f1) limit 1) and not exists
  (select * from P_S where old.k1 = p.k1 limit 1)
then   update C_S set f1 =null, f2 =null, f3 =null
  where ((f2 is null or f3 is null) and old.k1 = f1);
end if;
If   exists(select * from C_S where (f1 is null and
  f3 is null and old.k2 = f2) limit 1) and not exists
  (select * from P_S where old.k2 = p.k2 limit 1)
then   update C_S set f1 =null, f2 =null, f3 =null
  where ((f1 is null or f3 is null) and old.k2 = f2);
end if;
If   exists (select * from C_S where (f1 is null and f2
  is null and old.k3 = f3) limit 1) and not exists
  (select * from P_S where old.k3 = p.k3 limit 1)
then   update C_S set f1 =null, f2 =null, f3 =null
  where ((f1 is null or f2 is null) and old.k3 = f3);
end if; /* similar for all 2^n − 1 possible states */
End;
```

6.2 Index Structures

One feature that significantly affects the system behavior is the index structure applied to the referenced and referencing tables. An appropriate index structure can optimize searches and improve the performance by several reads in one scan.

0) *No*: No index is defined. This option is a baseline for judging the performance of actual indices.

1) *Full*: One index is defined on $[k_1, \dots, k_n]$ over P_S, and one index on $[f_1, \dots, f_n]$ over C_S. Full enforces simple semantics [22]. Full might not improve partial referential integrity enforcement since a null marker in the foreign key may lead to a complete scan on all parent key values from the leftmost to the rightmost column [9].

2) *Singleton*: One index is defined for each k_i over P_S, and for each f_i over C_S, for $i = 1, \dots, n$, resulting in $2n$ indices. With individual access to each column this approach is expected to boost the performance of enforcing partially-defined foreign keys [9].

3) *Hybrid*: One index is defined for each k_i over P_S, and a single index on $[f_1, \dots, f_n]$ over C_S, resulting in $n+1$ indices. According to [9], Hybrid takes advantage of both Full and Singleton options, and the authors conjectured that Hybrid best supports partial semantic.

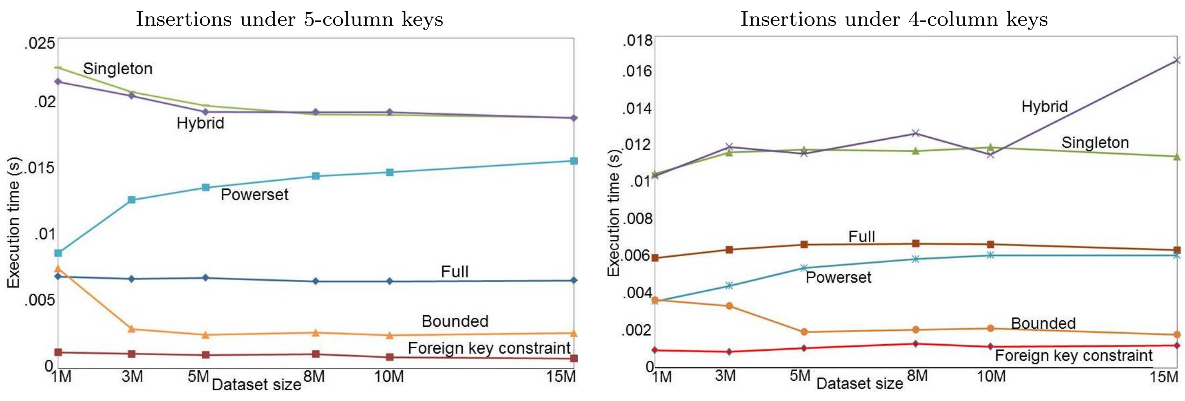
4) *Powerset*: One index is defined on each non-empty subset of P_S, and on each non-empty subset of C_S, resulting in $2^{n+1} - 2$ indices. Powerset shows the impact of having all possible indices available.

5) *Bounded*: One index is defined on $[k_1, \dots, k_n]$ and one index for each k_i over P_S, and one index is defined on $[f_1, \dots, f_n]$ and one index for each f_i over C_S, for $i = 1, \dots, n$, resulting in $2n + 2$ indices. This structure combines Full, Singleton, and Hybrid, and reduces Powerset to just the singletons (lower bound) and the full subset (upper bound) instead of all subsets. Bounded outperforms all other structures in our experiments.

Table 1: Execution Time (s) for Insertion with a 5-Column Foreign Key

Data Set Size	Partial Semantics												Simple Semantics	
	No Index		Full		Singleton		Hybrid		Powerset		Bounded			
	Ave	Max	Ave	Max	Ave	Max	Ave	Max	Ave	Max	Ave	Max	Ave	Max
15M	1.98	9.11	0.0066	0.109	0.019	0.187	0.0189	0.156	0.0157	0.312	0.0026	0.078	0.00076	0.046
10M	0.69	6.63	0.0065	0.093	0.019	0.188	0.0194	0.187	0.0148	0.343	0.0025	0.047	0.00085	0.031
8M	0.57	5.21	0.0066	0.094	0.019	0.187	0.0194	0.218	0.0145	0.312	0.0027	0.063	0.00109	0.031
5M	0.33	2.55	0.0068	0.078	0.019	0.172	0.0194	0.187	0.0137	0.125	0.0025	0.063	0.00103	0.031
3M	0.24	1.51	0.0067	0.094	0.021	0.234	0.0206	0.203	0.0127	0.297	0.0029	0.078	0.00110	0.047
1M	0.15	0.57	0.0069	0.078	0.022	0.156	0.0217	0.156	0.0087	0.093	0.0075	0.125	0.00123	0.016

Figure 4: Performance Trends for Enforcing Partial Semantics under Insertions and Different Indices



7. EXPERIMENTS ON SYNTHETIC DATA

We report on some of our experiments to evaluate the performance of enforcing partial semantics. Experiments were run on a Dell Latitude E5530, Intel core i7, CPU 2.9GHz with 8GB RAM. The operating system was Windows 7 Professional, Service pack 1 on a 64-bit operating system. The DBMS we used was MySQL version 5.6.

7.1 Description

All experiments involved two table schemata P_S and C_S and the foreign key $[f_1, \ldots, f_n] \subseteq P_S[k_1, \ldots, k_n]$ on C_S. Here, n varied between 2 to 5 to focus on the constraints that mostly occur in practice. For $n = 1$ there is no difference between simple and partial semantics. Parent table P and child table C were populated with synthetic data sets of various sizes between 1M and 15M tuples in P and 1.5 times as many tuples in C, respectively. Columns of the candidate key $\{k_1, \ldots, k_n\}$ on P_S did not feature `null`, while null markers did occur in the foreign key columns $\{f_1, \ldots, f_n\}$ in C. This allows us to gain insights on the foreign keys that mostly occur in practice. Each non-empty subset S of $\{f_1, \ldots, f_n\}$ had the same number of tuples in C which featured null markers in all columns in S and no null markers in any column outside of S. This also meant that the order of columns in a compound index does not matter in the experiments. We also run experiments where 50% and 80% of the tuples in C featured null markers in the foreign key columns, but the performances were very similar in each case. The performance of enforcing $[f_1, \ldots, f_n] \subseteq P_S[k_1, \ldots, k_n]$ was measured as the average time to insert tuples in C and to delete tuples from P, respectively, exploiting the triggers and different index structures from Section 6. For each data set and each index structure, the average was taken over 5,000 deletes and 5,000 inserts, respectively. We also compared the performance against that of simple semantics, enforced by built-in foreign keys. Note that execution times include the time for the trigger and the maintenance of the index structure. All reported experiments used BTrees. Applying Hash indices to our experiments resulted in similar outcomes, showing worse performance with minor exceptions. For these reasons we do not further comment on Hash indices here.

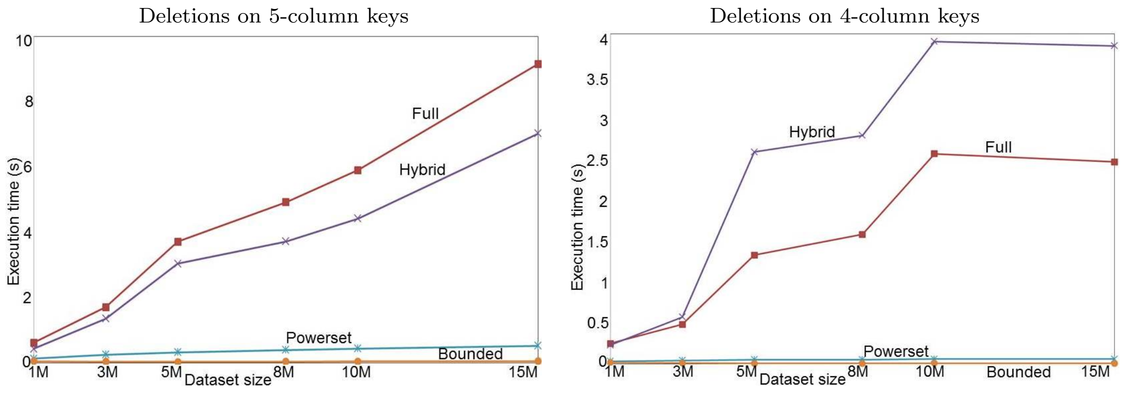
7.2 Impact of Indices

The impact of the index structures from Section 6 on the performance of enforcing partial semantics is the central contribution of our work. Tables 1 and 2 show the times to perform insertions into C and deletions from P, respectively, on the different data sets and where $n = 5$. These times are illustrated in Figures 4 and 5, respectively, along with the results for the same tests where $n = 4$. As expected, the use of indices leads to tremendous time savings for insertions and deletions. Our experiments confirm Härder and Reinhart's calculations that *Hybrid* achieves a performance similar to that of *Singleton* under insertions, and to that of *Full* under deletions. That is, it combines the performance gains of *Singleton* over *Full* under insertions, and the gains of *Full* over *Singleton* under deletions [9]. However, *Powerset* performs better than *Hybrid* under both insertions and deletions, and *Bounded* is the clear winner for both operations. On the largest data set with the largest foreign key size, for example, *Bounded* performs insertions/deletions on average about 7/123 times faster than *Hybrid*. The difference is considerable: the average time for deletions is 7.03s for *Hybrid* while it is 57ms for *Bounded*. *Bounded* is 6/9 times faster than *Powerset* on the largest data set. The

Table 2: Execution Time (s) for Deletion with a 5-Column Foreign Key

Data Set Size	Partial Semantics												Simple Semantics	
	No Index		Full		Singleton		Hybrid		Powerset		Bounded			
	Ave	Max	Ave	Max	Ave	Max	Ave	Max	Ave	Max	Ave	Max	Ave	Max
15M	286.04	824.7	9.16	203.1	110.45	813.9	7.03	142.20	0.531	0.967	0.057	0.296	0.0026	0.047
10M	201.38	480.3	5.91	151.3	109.00	651.7	4.42	100.34	0.442	0.904	0.047	0.234	0.0016	0.046
8M	128.04	393.7	4.93	119.8	93.32	572.8	3.72	82.41	0.401	0.92	0.042	0.218	0.0017	0.062
5M	85.79	255.48	3.7	86.15	39.2	337.1	3.04	67.76	0.330	0.79	0.037	0.188	0.0015	0.047
3M	52.65	138.76	1.7	63.43	15.4	143.3	1.36	41.52	0.255	0.655	0.041	0.156	0.0015	0.047
1M	19.61	41.77	0.62	13.58	3.5	25.4	0.44	8.70	0.140	0.577	0.057	0.266	0.0011	0.031

Figure 5: Performance Trends for Enforcing Partial Semantics under Deletions and Different Indices



performance gain of *Bounded* over *Powerset* shows that the additional time for maintaining further indices in *Powerset* and to choose the index from all the options in *Powerset* outweighs the time gains for the actual operations by *Powerset* in comparison to *Bounded.* Due to space restrictions we only present the results for 5-column foreign keys. For 3-column and 4-column foreign keys the index structures show similar behavior as the ones presented for 5-column foreign keys. Table 3 illustrates the results of applying *Bounded* and *Hybrid* on a synthetic data set of size 100M with a 5-column foreign key.

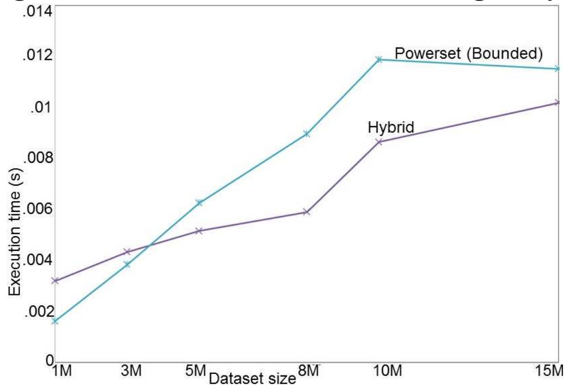
Table 3: Execution Time (s) for 100M Data Set under some Index Structures and 5-Column Foreign Key

	Insertion		Deletion	
	Mean	Max.	Mean	Max.
Hybrid	0.013	0.156	39.74	976.23
Bounded	0.0027	0.063	0.085	0.281
Simple S.	0.001	0.047	0.0021	0.062

Exception. *Hybrid* shows the best performance when $n = 2$ and the data set size is large. For example, on the largest data set it takes 2.8/10.2ms for insertions/deletions, while these times increase to 4.3/11.5ms on *Powerset*, see Figure 6. Note that *Powerset* and *Bounded* coincide when $n = 2$.

Simple semantics. Of course, it takes longer to enforce partial than simple semantics. However, the additional time becomes feasible under *Bounded.* For example, for insertions/deletions on the 15M data set with a 5-column foreign key, partial semantics is enforced by about 2.6/57ms, respectively. This takes 3.4/22 times longer than for simple

Figure 6: Deletions for 2-Column Foreign Keys



semantics, while it takes 22/2703 times longer than for simple semantics using *Hybrid.* Even on a data set with 100M tuples, inserts and deletions can be processed within 2.7ms and 84.8ms, respectively, using *Bounded.* This confirms the feasibility of enforcing partial semantics on even large foreign key sizes.

7.3 Index Building

While the time to load data and build the indices is just a *one-time* cost, we still include it in our analysis. Table 4 shows the time taken to load data and build the indices on all data sets. Not surprisingly, the more indices are defined the longer it takes to build them. Building *Powerset* is thus time-consuming: more than 3hrs and 53mins on the largest data set, while *Hybrid* takes just over 10mins. *Bounded*, with twice as many indices as *Hybrid*, takes about 14mins

Table 4: Times (s) for Loading Data and Building Indices for 5-Column Key

Data Size	No Index		Full		Singleton		Hybrid		Powerset		Bounded		Foreign key
	C	*P*	*C*	*P*	*C*	*P*	*C*	*P*	*C*	*P*	*C*	*P*	
15M	107.2	69.1	403.7	96.3	297.2	198.3	405.8	200.9	5269.2	8716.8	622.6	234.5	587.9
10M	69.0	43.9	234.5	63.8	199.2	132.9	242.6	134.7	2875.4	5305.8	366.8	156.7	262.0
8M	56.4	36.0	172.3	51.8	159.3	107.6	171.7	106.0	2061.3	4073.4	275.5	120.5	180.3
5M	42.2	22.8	99.3	32.2	95.2	67.0	94.2	63.9	1016.3	2298.1	160.5	77.6	100.9
3M	20.5	13.3	53.5	19.5	63.9	41.4	54.0	40.7	522.6	1162.0	96.2	47.1	53.6
1M	7.9	4.9	16.7	7.4	20.1	13.6	15.3	13.3	142.6	151.1	28.2	14.72	16.9

and 30s. The foreign key index is built in 9mins and 47s.

7.4 Impact of Update Size

We performed some experiments with transactions, that is, atomic sets of update operations. The reported experiments were conducted for the 5-column foreign key on the data set with 15M tuples. The first experiment featured an insert of 5,000 tuples into C, and the second experiment a deletion of 2,000 tuples from P. The results are illustrated in Table 5. The transaction with 5,000 inserts takes just under 7s with *Bounded*, and nearly 90s with *Hybrid*. For the transaction with 2,000 deletions it takes over 148mins with *Hybrid*, and just under 111s with *Bounded*.

Table 5: Execution Time (s) of Transactions under Index Structures on Data Set with 15M Tuples

	5000 Insertions	2000 Deletions
Full	28.533	13413.16
Singleton	90.137	59191.81
Hybrid	89.65	8922.6
Powerset	102.81	605.71
Bounded	6.973	110.37
Simple S.	0.811	32.92

7.5 Extended Tests and Analysis

Deletions. Tables 2 and 5 show the poor performance of *Hybrid* on delete actions, and that it is overcome by *Bounded*. We will now analyze how *Bounded* achieves this. For that purpose, we exploited MySQL's explain statement which shows the optimizer's plan in executing the statements of each test. Recall that *Hybrid* has only one compound index over all foreign key columns in C_S. When we *Delete* from P, this means that one scan through all tuples is required to apply the referential action to children that feature `null` on the left most column. However, the referential action is only needed when there is no alternative parent for these children. Establishing that referential action must be applied to children whose only parent has been deleted is therefore poorly supported by *Hybrid*. We validated this observation by applying a deeper analysis to the experiment with our data set of 10M tuples. For this purpose, we call a parent *unique* when it has only children for which it is the only parent. Therefore, referential actions apply to all children of a unique parent. Otherwise, the parent is called *non-unique*. Table 6 shows the average execution time for deleting unique and non-unique parents when *Hybrid* is applied.

Table 7 shows the average execution time for deleting non-unique and unique parents when *Bounded* is applied.

These results show that *Hybrid* performs particularly poor when deleting unique parents. The same analysis applies to transactions. In fact, only 3% of the deleted parents were unique. With *Hybrid* they occupied 145mins of the overall time of 148mins, but with *Bounded* they occupied only 0.5s of the overall time of 111s, refer to Table 13.

Table 6: *Hybrid* for Deletions, Average of Execution Times

Key size	Non-unique Parent	Unique Parent
5-Column keys	0.525s	40.47s
4-Column keys	2.37s	18.68s
3-Column keys	0.022s	17.59s

Table 7: *Bounded* for Deletions, Average of Execution Times

Key size	Non-Unique Parent	Unique Parent
5-Column keys	0.051s	0.005522s
4-Column keys	0.00976s	0.00305s
3-Column keys	0.00348s	0.00167s

This poor performance of *Hybrid* can be avoided by adding to *Hybrid* one index on each foreign key attribute f_i on C_s. The resulting structure consists of $2n + 1$ indices in total, and we refer to it by *Hybrid+nSingle*. Table 8 shows the average execution time for deleting non-unique and unique parents when *Hybrid+nSingle* is applied.

Table 8: *Hybrid+nSingle* for Deletions, Average of Execution Time

Key size	Non-Unique Parent	Unique Parent
5-Column keys	0.413s	0.0061s
4-Column keys	2.29s	0.003s
3-Column keys	0.0237s	0.00233s

Another index structure that we have also tested is *Hybrid+Compound*, which consists of *Hybrid* plus one index over the key columns of P_S. *Hybrid+Compound* has therefore a total of $n + 2$ indices. For deletions, the additional index improves the search for children which are not null in the leftmost columns. This is demonstrated by comparing the results for *Non-Unique Parents* in Tables 7 and 8.

Figure 7 illustrates how well *Hybrid+nSingle* and *Hybrid+Compound* perform deletions in comparison to the other indices. Clearly, the performance boost of *Bounded* over *Hybrid* for deletions is mainly due to adding *nSingle*.

Insertions. Figure 8 illustrates how well *Hybrid+nSingle* and *Hybrid+Compound* perform insertions in comparison to the other indices. Clearly, the performance boost of *Bounded* over *Hybrid* for insertions is mainly due to adding *Compound*.

A deeper analysis confirms our intuition that *Hybrid* performs particularly poorly when inserting tuples that have only total foreign key values. Figure 9 breaks down the per-

Figure 7: Performances under Deletions with 5-Column Foreign Keys

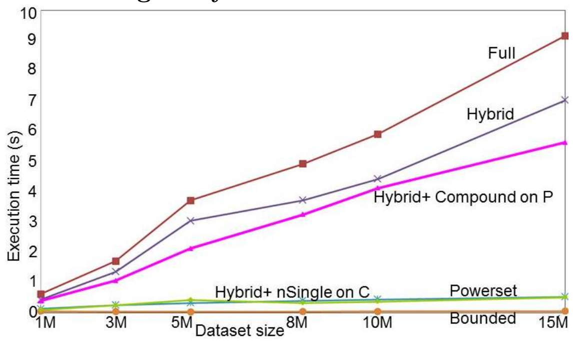
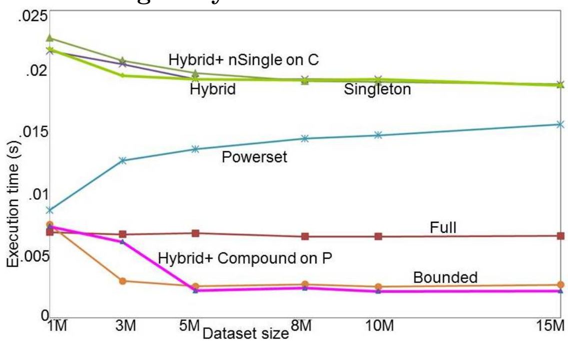
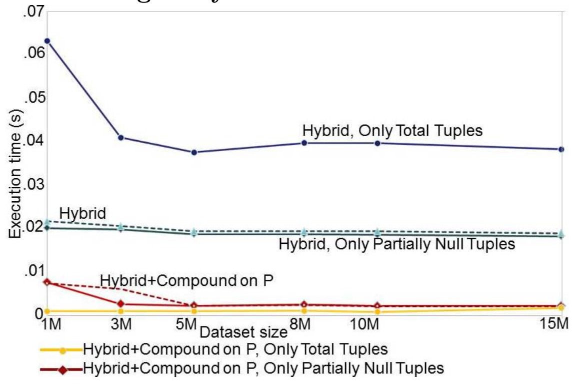


Figure 8: Performance under Insertions with 5-Column Foreign Keys



formance of *Hybrid* and *Hybrid+Compound* into insertions of tuples with only total foreign keys and those that are partially `null`.

Figure 9: Performance under Insertions with 5-Column Foreign Keys



Bounded is the only index structure that performs well under insertions and deletions, since it does not suffer from poor performance for insertions like *Hybrid+nSingle* and for deletions like *Hybrid+Compound*. In comparison to *Hybrid+nSingle* the better performances are achieved under negligible additional costs for building the indices, see Tables 11 and 12.

Transactions. Table 13 shows how the new index structures perform in transactions. *Hybrid+Compound* performs best for insertions but takes 149mins for deleting 2000 parents, while *Bounded* takes just 110s. *Hybrid+nSingle* is the runner-up to *Bounded* for deletions, but performs poorly for insertions.

Table 11: Index Building (IB) and Execution Time of *Bounded*

Dataset Size	IB for C (s)	IB for P (s)	Insert Ave. (s)	Delete Ave. (s)
15M	622.569	234.532	0.002678	0.0578
10M	366.821	156.73	0.00251	0.047189
8M	275.513	120.542	0.00271	0.0425
5M	160.463	77.563	0.00254	0.03723
3M	96.237	47.019	0.002988	0.04189
1M	28.173	14.742	0.00757	0.05729

Table 12: Index Building (IB) and Execution Time of *Hybrid+nSingle*

Dataset Size	IB for C (s)	IB for P (s)	Insert Ave. (s)	Delete Ave. (s)
15M	636.702	207.029	0.0189	0.5135
10M	367	136.204	0.0194	0.37
8M	282.409	109.045	0.01934	0.3316
5M	162.6	66.81	0.0194	0.43
3M	92.212	40.68	0.0197	0.2481
1M	28.25	13.26	0.02186	0.10149

Table 13: Execution Time (s) of Transactions under Index Structures

	5000 Insertions	2000 Deletions	
		Unique p	Others
Hybrid	89.65	8753.2	169.4
Hybrid+Compound	6.26	8830.8	104.05
Hybrid+nSingle	90.78	0.515	167.43
Bounded	6.973	0.499	109.9

8. BENCHMARK DATA

We extend our analysis of enforcing partial semantics to some benchmark and real-world data. For that purpose, we have run experiments on one two-column foreign key from the TPC-H database and two three-column foreign keys from the TPC-C database (`www.tpc.org`). In addition, we have tested one three-column foreign key from the Gene Ontology (GO) database (`www.geneontology.org/GO.database.shtml`). Table 9 shows the details of the foreign keys. Here, the data set size for test 1 on TPC-H was 1.43GB, and for test 2 it was 10GB; for TPC-C it was 0.39GB, and for the GO database it was 100MB. Applying the "Missing at Random" mechanism from [23], null markers were introduced randomly and spread evenly between the foreign key columns.

We have tested the TPC-H benchmark with two different data set sizes (0.8M and 8M tuples). Note that *Powerset* and *Bounded* coincide on 2-column foreign keys and thus on the experiments with TPC-H. The results of enforcing partial and simple semantics on these databases are shown in Table 10. The performances rank very similar to those on the synthetic data sets. Our results on TPC-H confirm the observed changes on the performance of *Hybrid* and *Powerset* on larger data sets with 2-column foreign keys, see Figure 6. The TPC-C data set with the 3-column foreign keys confirms our result with the synthetic data sets that

Table 9: Detail of the tested TPC databases

Database	Parent table	Child table	Foreign key
TPC-H	Partsupp Test 1: 0.8M records Test 2: 8M records	Lineitem 6M records 25% Null 60M records 60% Null	[*partkey*, *suppkey*] ⊆ Partsupp[*partkey*, *suppkey*]
TPC-C	Customer (90K records)	Orders (0.13M records) 55% Null	[*O-W-ID*, *O-D-ID*, *O-C-ID*] ⊆ Customer[*C-W-ID*, *C-D-ID*, *C-ID*]
TPC-C	Orders (0.13M records)	Orderline (1.3M records) 20% Null	[*OL-W-ID*, *OL-D-ID*, *OL-C-ID*] ⊆ Orders[*O-W-ID*, *O-D-ID*, *O-ID*]
Gene Ontology (GO) database	Term2term (T) (80k records)	Term2term-metadata (TT) (2200 records) 85% Null	[*relationship-type-id*, *term1-id* ,*term2-id*] ⊆ [*relationship-type-id* , *term1-id*, *term2-id*]

Table 10: Execution Time(ms) to Enforce Partial Referential Integrity on Benchmark Databases

	No Index	Full	Singleton	Hybrid	Powerset	Bounded	Simple S.
TPC-H Test 1: Insert into Lineitem	161	1.8	1.6	1.3	1.1	-	1.06
Delete from Partsupp	10.92 (s)	6.1	148	5.6	4.3	-	2
TPC-H Test 2: Insert into Lineitem	1.97s	3.1	1.1	0.76	1.09	-	0.88
Delete from Partsupp		19.7	151.72	5.94	14.79	-	3.10
TPC-C: Insert into Orders	14.5	2.73	2.95	2.84	1.28	1.02	0.81
Delete from Customer	498	11.8	120.3	10.15	2.44	2.47	0.6
TPC-C: Insert into Orderline	9.23	2.44	1.9	2.25	1.62	1.31	1.21
Delete from Orders	2.92(s)	18.6	1.05(s)	15.58	3.78	3.52	1.285
GO Database: Insert into TT	15.6	6.16	1.31	1.16	2.37	1.3	1.12
Delete from T	54.2	2.11	17.35	12.6	2.4	1.87	1.06

Bounded outperforms *Hybrid* by a factor of 2 for insertions and by 5 for deletions. Similar results have been observed on the Gene Ontology database. The best times for enforcing partial semantics are never over 6ms while the times for enforcing simple semantics are never over 4ms.

9. CONCLUSION AND FUTURE WORK

Even two decades after its introduction to the SQL standard there are no database management systems with built-in support for partial referential integrity. Härder and Reinhart had argued on the operational level that partial semantics is too costly to implement [9]. They invited more research on its motivation and its costs at the systems level [9]. In this paper we addressed both questions. Firstly, we proposed intelligent update and query services that impute missing data values by exploiting partial semantics. In particular, we discussed how these services reduce information incompleteness in the database and suggest possible values by which null markers can be replaced in query answers. Secondly, we proposed two main operational requirements to enforce and speed up the enforcement of partial semantics, and conducted several performance tests in MySQL. Our tests were targeted at foreign keys that occur commonly in practice. These have very rarely more than five columns and reference candidate keys without null marker occurrences. Permitting occurrences of `null` in referenced candidate keys only affects our results marginally. Our first major finding confirms on the system level what Härder and Reinhart had calculated on the operational level: (1) The performance of the *Hybrid* index structure matches that of the *Singleton* index structure for insertions, and that of the *Full* index structure for deletions. (2) *Hybrid* is the best candidate to support `MATCH PARTIAL`, but only for 2-column foreign keys. We proposed here the new index structure *Bounded* for foreign keys with more than two columns. Our second major finding is that *Bounded* outperforms *Hybrid* for foreign keys with more than two columns. For example, for a 5-column foreign key on a data set with 15M tuples and a fair distribution of null markers, *Bounded* performed 7 times better for insertions and 123 times better for deletions. The better performance was confirmed on other synthetic data sets, for transactions, for two TPC-C data sets and a three-column foreign key on the Gene Ontology database. The only trade-off we found concerns the loading time of the data set and building of the indices, which essentially were 1.5 times more in comparison to *Hybrid*. Our third major finding is that the performance boost of *Bounded* for deletions results from adding one index to *Hybrid* on each foreign key column, and the performance boost of *Bounded* for insertions results from adding one index on the compound key to *Hybrid*. Overall, the enforcement of partial semantics with *Bounded* was never slower than 300ms for any atomic operation with a 5-column foreign key, while the enforcement of simple semantics was never slower then 62ms. Based on our results, we conclude that partial referential integrity can be enforced efficiently for real-life foreign keys, which opens up new applications such as intelligent queries and intelligent updates that lead to better quality data and better data-driven decision making. These applications do not apply to simple referential integrity. Our results demonstrate the benefits of partial semantics, and *Bounded* offers a principled approach to indexing foreign keys with partial semantics that can be added on top of existing databases in a non-intrusive fashion.

We hope our research will ignite future work on this topic. Certainly there are many interesting questions that a single paper cannot address, but which should be pursued in future work to unlock many potential benefits. Other indexing options can be studied, for example the combination of compound indices over key attributes and multidimensional access paths at the system level. An index option including $2n$ compound n-ary indices over the referenced key attributes $[k_i, \ldots, k_n, k_1, \ldots, k_{i-1}]$ and referencing key attributes $[f_i, \ldots, f_n, f_1, \ldots, f_{i-1}]$, for $i = 1, \ldots, n$, supports partial match index look-ups by the different prefixes of the compound indices. However, our initial analysis shows that *Bounded* outperforms this index option in deletions of 3, 4 and 5-column foreign keys by more than 3 times on data sets with 15M tuples. The loading times of the data sets and building of the indices in *Bounded* are always between 1.5 to 4 times cheaper. Exploiting $2n$ compound indices over key attributes is not enough to support all the possible partial match queries. For instance when $n = 5$, defining 2×5 compound indices in different orders only supports 21 of 31 match queries. Another interesting avenue deals with the trade-off between query and enforcement issues. While our results show that enforcement and maintenance of partial semantics are unlikely bottlenecks on databases of enterprise-level size (10GB in test 2 for TPC-H), future research should be aimed at guidelines for resolving conflicts between resources for query and update acceleration in schemata for big data. While our solution of implementing enforcement by database triggers and index primitives is appealing in several ways, future work may reveal potential performance gains that could be realized with an engine-level implementation. For instance, there may be custom index data structures that leverage partial and adaptive indexing methods, as well as a more streamlined trigger execution in order to improve enforcement costs. Furthermore, there are several techniques such as batching and shared execution across updates that apply within transactions, and could therefore optimize the enforcement of partial referential integrity in this context. Our proposed services of intelligent queries and updates each open up their own areas of future investigation. For updates it would be interesting to investigate how large numbers of choices for imputations can be represented or ranked, how logs of potential imputations can best be processed by data analysts, or how unsuccessful imputations can be reversed. For queries, it would be interesting to find re-writings of SQL queries that return not only standard but also non-standard answers that result from the application of partial semantics, and to investigate the overhead of such techniques. Information incompleteness is also inherent in other data models such as graphs, RDF, or XML. Finally, implication problems of inclusion dependencies under SQL semantics should be studied [10, 11].

10. REFERENCES

[1] J. Bauckmann, Z. Abedjan, U. Leser, H. Müller, and F. Naumann. Discovering conditional inclusion dependencies. In X. Chen, G. Lebanon, H. Wang, and M. J. Zaki, editors, *CIKM*, pages 2094–2098. ACM, 2012.

[2] M. A. Casanova, R. Fagin, and C. H. Papadimitriou. Inclusion dependencies and their interaction with functional dependencies. *J. Comput. Syst. Sci.*, 28(1):29–59, 1984.

[3] M. A. Casanova, L. Tucherman, and A. L. Furtado. Enforcing inclusion dependencies and referential integrity. In F. Bancilhon and D. J. DeWitt, editors, *VLDB*, pages 38–49. Morgan Kaufmann, 1988.

[4] A. K. Chandra and M. Y. Vardi. The implication problem for functional and inclusion dependencies is undecidable. *SIAM J. Comput.*, 14(3):671–677, 1985.

[5] E. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[6] C. J. Date. *Relational database writings: 1985-1989.* Addison-Wesley, 1990.

[7] V. Ganti and A. D. Sarma. *Data Cleaning: A Practical Perspective.* Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2013.

[8] T. Halpin and T. Morgan. *Information modeling and relational databases.* Morgan Kaufmann, 2010.

[9] T. Härder and J. Reinert. Access path support for referential integrity in SQL2. *The VLDB Journal*, 5(3):196–214, 1996.

[10] S. Hartmann, M. Kirchberg, and S. Link. Design by example for SQL table definitions with functional dependencies. *The VLDB Journal*, 21(1):121–144, 2012.

[11] S. Hartmann and S. Link. The implication problem of data dependencies over SQL table definitions. *ACM Trans. Database Syst.*, 37(2):13, 2012.

[12] D. S. Johnson and A. C. Klug. Testing containment of conjunctive queries under functional and inclusion dependencies. *J. Comput. Syst. Sci.*, 28(1):167–189, 1984.

[13] M. Karlinger, M. W. Vincent, and M. Schrefl. Inclusion dependencies in XML: Extending relational semantics. In S. S. Bhowmick, J. Küng, and R. Wagner, editors, *DEXA*, volume 5690 of *LNCS*, pages 23–37. Springer, 2009.

[14] G. Lausen. Relational databases in RDF: Keys and foreign keys. In V. Christophides, M. Collard, and C. Gutierrez, editors, *SWDB-ODBIS*, volume 5005 of *LNCS*, pages 43–56. Springer, 2008.

[15] M. Levene and M. W. Vincent. Justification for inclusion dependency normal form. *IEEE Trans. Knowl. Data Eng.*, 12(2):281–291, 2000.

[16] S. Link and M. Memari. Static analysis of partial referential integrity for better quality SQL data. In J. Shim, Y. Hwang, and S. Petter, editors, *AMCIS*, pages 38–49. Association for Information Systems, 2013.

[17] J. Melton. ISO/IEC 9075-2: 2003 (SQL/foundation). ISO standard, 2003.

[18] J. C. Mitchell. The implication problem for functional and inclusion dependencies. *Information and Control*, 56(3):154–173, 1983.

[19] J. C. Mitchell. Inference rules for functional and inclusion dependencies. In R. Fagin and P. A. Bernstein, editors, *PODS*, pages 58–69. ACM, 1983.

[20] C. Ordonez and J. García-García. Referential integrity quality metrics. *Decision Support Systems*, 44(2):495–508, 2008.

[21] B. Thalheim. *Dependencies in relational databases.* Teubner, 1991.

[22] C. Türker and M. Gertz. Semantic integrity support in SQL: 1999 and commercial (object-) relational DBMSs. *The VLDB Journal*, 10(4):241–269, 2001.

[23] Y. Wen, K. B. Korb, and A. E. Nicholson. DataZapper: Generating incomplete datasets. In J. Filipe, A. L. N. Fred, and B. Sharp, editors, *ICAART*, pages 69–76. INSTICC Press, 2009.

[24] M. Zhang, M. Hadjieleftheriou, B. C. Ooi, C. M. Procopiuc, and D. Srivastava. On multi-column foreign key discovery. *PVLDB*, 3(1):805–814, 2010.

Query Optimization over Cloud Data Market

Yu Li, Eric Lo, Man Lung Yiu, Wenjian Xu
Department of Computing
Hong Kong Polytechnic University
{csyuli, ericlo, csmlyiu, cswxu}@comp.polyu.edu.hk

ABSTRACT

Data market is an emerging type of cloud service that enables a data owner to sell their data sets in a public cloud. Buyers who are interested in a certain dataset can access the data in the market via a RESTful API. Accessing data in the data market may not be free. For example, it costs USD 12 per month to obtain 100 "transactions" from the WorldWide Historical Weather dataset in Windows Azure Data Marketplace, where a transaction is a unit of result size (e.g., a query result of 4400 records would consume 44 transactions as Windows Azure Data Marketplace confines one transaction to 100 records). Therefore, in this paper, we present PayLess, a system that helps data buyers to optimize their queries so that they can obtain the query results by paying less to the data sellers. Experiments over synthetic data and real data sets in Windows Azure Marketplace show that PayLess can cost-effectively handle SQL query processing over data markets.

1. INTRODUCTION

Data market [1, 16, 42] is an emerging type of cloud service that enables a data owner to host and sell their datasets in a public cloud. Buyers who are interested in a certain dataset can access the data in the market via a RESTful API. The REST based API has function-call like interface $\mathcal{X} \to \mathcal{Y}$, where $\mathcal{X}$ and $\mathcal{Y}$ are sets of attributes: given a range or a value for an attribute in $\mathcal{X}$, the data market returns values for the attributes in $\mathcal{Y}$ (if no values are specified for $\mathcal{X}$, the whole table is returned). For example, the Worldwide Historical Weather (WHW) dataset [13] in Windows Azure Marketplace [1] may take a country name and a date, and return a set of tuples, each details the temperature, precipitation, dew point, sea level pressure, windspeed, and wind gust recorded by each weather station in that country on that date.

Accessing data in the data market may not be free. For example, it costs USD 12 to grant access to every 100 "transactions" to the WHW data, where a transaction is a unit of result size (e.g., a query result of 4400 records costs 44 transactions in Windows Azure Marketplace, which confines one transaction to 100 records). There is an increasing trend of selling valuable datasets in data market [31]. Correspondingly, we envision that there is an increasing demand from end users (data buyers) to carry out analytics that involve those datasets. To this end, in this paper, we present PayLess, a system that helps users to optimize their queries so that they can obtain the query results by *paying less* to the data sellers.

Query optimization is never trivial. First, from a data buyer's (the company or the organization) perspective, it is hard to know in advance how many queries will be posed by their end users eventually. Otherwise, downloading the whole dataset would become a viable plan when the foreknowledge tells that the number of transactions incurred by user queries would eventually exceed the number of transactions required to download the complete data set. Second, query optimization would never work well without rich data statistics. Unfortunately, datasets in data market are rarely tagged with rich statistics (e.g., no value distribution), although basic information like the size (cardinality) of each table and the domain size of the attribute is usually available.

Tackling the above two challenges sounds not difficult, especially that we can build a *learning* optimizer like LEO [46] so that it begins with little statistic and introduces a feedback loop to correct the statistics when more queries are issued. The evil, however, lies in the detail of adopting the learning approach to data market query optimization.

First, learning-based optimizers like LEO [46] and POP [38] are originally designed for traditional databases that have full access to the data. In contrast, the access pattern of data market is restricted to only $\mathcal{X} \to \mathcal{Y}$ style. When a data source has limited access patterns, (a) operations might become complicated and (b) specialized access paths may shine. An example of (a) is that a query that asks `Country = 'Canada' OR Country = 'Germany'` has to decompose into two queries, one asks for `Country = 'Canada'` and another asks for `Country = 'Germany'`. An example of (b) is *bind joins* (other names include theta semi-join, dependent join) [27]. To explain, consider the real access patterns of Worldwide Historical Weather (WHW) dataset in Windows Azure Marketplace listed in Figure 1a.[1] The access patterns are specified using a notation of binding patterns extended from [27]. We write $R^{\alpha}(A_1, A_2, A_3)$ to denote a table R in the data market with three attributes A_1, A_2, and A_3 and *binding pattern* α. We write $\alpha = R(A_1^b, A_2^f)$ to denote a binding pattern that in any query accessing R, the value of attribute A_1 must be *bound* (given/specified). In contrast, the value of attribute A_2 is *free* to be specified or not specified in any query. If an attribute is not included in the binding pattern (e.g., A_3), it is solely served as an output attribute in a query result. In other words, if an access pattern of a table has only free attributes, then we can download the whole table by not specifying any value to any attribute.

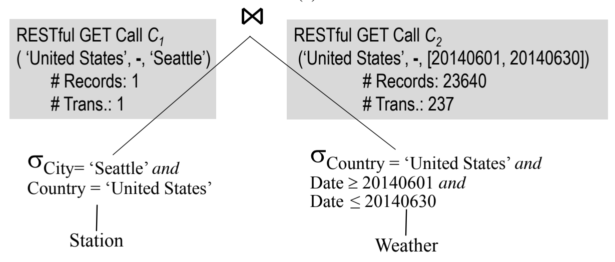
Now, consider the following SQL query that asks the WHW

[1]The attribute names here are renamed for better exposition.

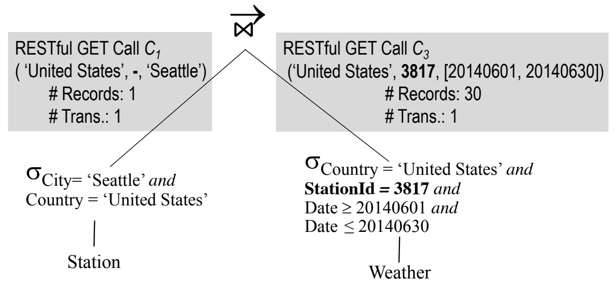
10.5441/002/edbt.2015.21

Data Set	Schema and Access Pattern α	Size
WHW	Station$^{\alpha_1}$(`Country, StationID, City, State···`) α_1=Station(`Country`^{f}, `StationID`^{f}, `City`^{f})	3962
	Weather$^{\alpha_2}$(`Country, StationID, Date, Temperature ···`) α_2=Weather(`Country`^{f}, `StationID`^{f}, `Date`^{f})	19549140
EHR	Pollution$^{\alpha_3}$(`ZipCode, Rank, Latitude, Longitude···`) α_3=Pollution(`ZipCode`^{f}, `Rank`^{f})	44210
local	ZipMap (`ZipCode, City`)	

(a)


⋈
RESTful GET Call C_1
('United States', -, 'Seattle')
# Records: 1
# Trans.: 1
RESTful GET Call C_2
('United States', -, [20140601, 20140630])
# Records: 23640
# Trans.: 237
σCity= 'Seattle' and Country = 'United States'
Station
σCountry = 'United States' and Date ≥ 20140601 and Date ≤ 20140630
Weather

(b) Plan P_1


RESTful GET Call C_1
('United States', -, 'Seattle')
# Records: 1
# Trans.: 1
RESTful GET Call C_3
('United States', 3817, [20140601, 20140630])
# Records: 30
# Trans.: 1
σCity= 'Seattle' and Country = 'United States'
Station
σCountry = 'United States' and StationId = 3817 and Date ≥ 20140601 and Date ≤ 20140630
Weather

(c) Plan P_2

Figure 1: Query Processing in Data Market

dataset for the daily temperature of Seattle in June 2014:

```
SELECT Temperature          -------// Query Q1
FROM Station, Weather
WHERE City = 'Seattle' AND
      Country = 'United States' AND
      Date >= 20140601 AND Date <= 20140630 AND
      Station.StationID = Weather.StationID
```

Figure 1b shows an execution plan P_1 for this SQL. It first submits two RESTful GET calls C_1 and C_2, where C_1 gets the StationID of Seattle from Station table, and C_2 gets the weather records for all stations in the United States on June 2014 from Weather table. The final query result is obtained by carrying out a *local join* (i.e., regular join) operation at the end user (data buyer) side because joins cannot be done at the data market [1]. In plan P_1, a total of 238 transactions were incurred – one was spent on RESTful call C_1 and 237 were spent on RESTful call C_2 (there are 788 weather stations in the US and each station contributes 30 days records, resulting in $\lceil 788 \times 30/100 \rceil = 237$ transactions). Figure 1c shows an alternate execution plan P_2. It first gets the list of StationIDs of Seattle (call C_1). Then, it carries out a bind join ($\overrightarrow{\bowtie}$) operation that binds each StationID (e.g., 3817) to an individual RESTful call to Weather. Finally, the weather records for each station in Seattle are collectively retrieved and returned. In this case, plan P_2 incurs only two transactions: call C_1 costs one transaction and call C_3, which returns 30 days of weather records for the only one weather station in Seattle, costs also one transaction.

Second, although there are optimizers designed for queries over remote data sources with limited access patterns (e.g., [17, 24, 27, 33–36, 40, 45]), they focus on minimizing the number of calls to the remote data sources so as to reduce the overall execution time. As an example, assume that there are 15 weather stations in Seattle, those optimizers will pick plan P_1 because it incurs only two RESTful calls (C_1 and C_2). In data market, although P_2 needs to bind each Seattle's weather station id, resulting in $1 + 15 = 16$ RESTful calls and 16 transactions (each transaction returns 30 days of records for each weather station), it is still more economy than P_1, which requires 238 transactions. On the other hand, if we further assume that there are only 20 weather stations in the United States and 15 of them are in Seattle. Then, plan P_1 will cost only $1+\lceil 20\times 30/100 \rceil = 7$ transactions. In contrast, plan P_2 still costs 16 transactions. In this case, P_1 is better than P_2.

Summing up the above, we need a (i) learning-based optimizer that (ii) includes bind join as an access path with the goal of (iii) minimizing the amount of (intermediate) retrieved data measured in terms of data market pricing units. Traditional learning-based optimizers satisfy (i) and partially satisfy (iii)[2] but not (ii). Optimizers for queries over remote data sources satisfy only (ii). Therefore, the principal contributions of this paper are centered around the issues of building an optimizer for PayLess that satisfies all (i), (ii), and (iii) above. Those include:

- Defining the cost model and search space for data market query optimization.
- Devising effective techniques to reduce the amount of intermediate retrieved data (e.g., by adapting semantic query rewriting methods) and integrating those techniques into our optimizer.
- Implementing a prototype and evaluating its performance through extensive experiments over synthetic data and real data.

The remainder of this paper is organized as follows. Section 2 gives more background about the data market. Section 3 presents the architecture of PayLess. Section 4 describes the details of PayLess's optimizer. Section 5 reports the results of the evaluation. Section 6 discusses the related work and Section 7 concludes.

2. PRELIMINARIES

According to a recent survey [2], the three most established data marketplaces are Factual [8], Microsoft Windows Azure Data Marketplace [1], and DataMarket [4]. Factual [8] and DataMarket [4] are specialized data markets that sell datasets in a very specific domain (e.g., Factual sells mainly geographical data and DataMarket sells mainly economic indicators). Microsoft Windows Azure Data Marketplace offers data sets in all kinds and many popular data resellers in smaller size like Wolfram Alpha [11], ESRI [7], World Bank [12], data.gov [3], Xignite [14] also provide their data in the Windows Azure Data Marketplace [1]. After Infochimps [9], one of the early data market entrants, gradually leaves the data market business [5, 10], Microsoft Windows Azure Data Marketplace is becoming the de facto data market [2]. Therefore, in this paper, we base our setting on Windows Azure Data Marketplace.

2.1 Data Market

A data market hosts and sells multiple datasets. Each dataset's access/binding pattern is defined by the data owner on per table basis. For numeric attributes, the input can be bound with a single value or a range like $[150, 200)$. Datasets in data market are tagged with very basic statistics, normally the domain of each attribute and

[2]Traditional optimizers also aim to generate plans that minimize intermediate result size of each operation (e.g., push down selection).

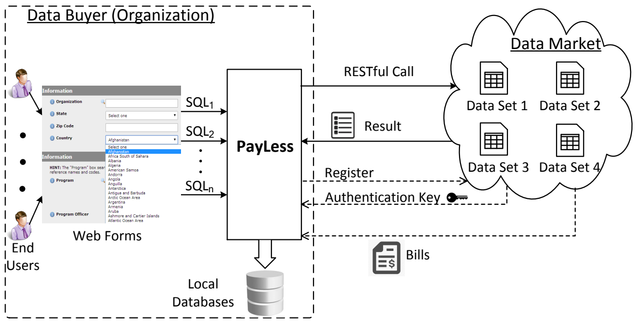


Figure 2: Setting of PayLess

the number of records (cardinality).[3] Datasets in a data market are *append-only* because they are released for analytic purposes. New data could be added periodically (e.g., every month). The price of accessing data is mainly based on the number of tuples retrieved. A *transaction* represents a page of t tuples (e.g., 100 tuples) and it is the smallest pricing unit. Let p be the price per transaction for a particular dataset. Then, the total price of a RESTful call is:

$$p \cdot \lceil \frac{\text{number of resulting records}}{\text{number of tuples per transaction } (t)} \rceil \quad (1)$$

For easy exposition, in the subsequent discussion, we assume $p =$ \$1 and a transaction page size is $t = 100$ tuples.

2.2 Queries over Data Market

Figure 2 shows the target setting of PayLess. An organization is interested in carrying out certain analytics that involve datasets hosted in a data market. The organization thus registers with the data market to obtain the authentication access keys of the datasets. The access keys are stored in PayLess, which constructs RESTful calls to the data market when necessary. PayLess encapsulates the details of interacting with the data market and exposes a SQL query interface for client query processing. A SQL query to PayLess can query against both tables in a local DBMS and tables in data market. The following is an example PayLess query that aims to retrieve the average temperature for each city in a country whose environmental pollution rank is lower than a threshold within a period:

```
SELECT City, AVG(Temperature)
FROM Pollution, Station, Weather, ZipMap
WHERE Station.Country = Weather.Country = ? AND
      Weather.Date >= ? AND Weather.Date <= ? AND
      Pollution.Rank <= ? AND
      Pollution.ZipCode = ZipMap.ZipCode AND
      ZipMap.City = Station.City AND
      Station.StationID = Weather.StationID
GROUP BY City
```

This query involves joining four tables: the Station and Weather tables from the aforementioned Worldwide Historical Weather (WHW) [13] dataset, another Data Market table, the Pollution table from the Environmental Hazard Ranking (EHR) [6] dataset, and a local table that maps Zip codes to a city name. The access patterns of these tables are shown in Figure 1a. We expect SQL queries to PayLess are parameterized queries embedded in certain application so that users (e.g., data scientists) issue the queries by specifying the parameter values via a web interface. We do not expect the organization restricts her users the number of queries to the data market because that is counter-productive.

[3]If not publicly available, the data sellers would release the basic statistic to data buyers upon email requests [1].

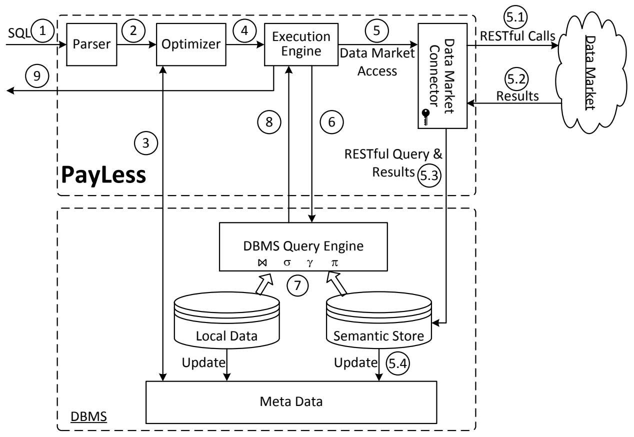


Figure 3: System Architecture

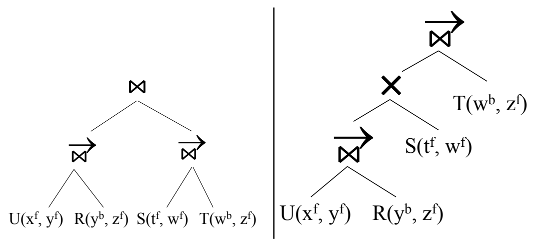
3. SYSTEM OVERVIEW

Figure 3 shows the architecture of PayLess. It is designed to be lightweight and offloads most query processing to a DBMS query engine. It accepts and parses a SQL query (with parameter values instantiated) ①. The parser differentiates local tables and tables from the data market using the information (e.g., the table name) obtained when registering with the data market (see Figure 2). Then, the optimizer of PayLess optimizes the query ② by consulting the statistics of local and data market data ③. The optimized query is then passed to an execution engine ④. A query, after optimization, may be able to skip some or the entire access to the data market. When it is necessary to access the data market, the execution engine will pass the access requests to the data market connector ⑤ and let the connector interact with the data market (5.1) (5.2). PayLess stores all the data market access requests and their returned data in a semantic store (5.3). Whenever new data is retrieved from the data market, PayLess will update its statistics (5.4). In our implementation, we implement our updatable statistics using ISOMER [44]. After this step, all data required by a query should be ready and stored in the DBMS and the execution engine of PayLess instructs the DBMS query engine ⑥ to process the query ⑦. In the end, the execution engine of PayLess retrieves the query result from the DBMS ⑧ and then returns it to the front end ⑨.

PayLess is supposed to be installed by each data buyer and serves all the end users from the same data buyer. As a data buyer would not be interested in all datasets available in the data market, the storage space (for the DBMS) is not a problem here. Cache management is out of PayLess's interest because we deliberately use cheap storage space to store all intermediate results (i.e., no eviction) in order to eschew retrieving redundant data from the data market. Besides, PayLess is indeed amenable for any updatable statistic. As our focus of this paper is to give a proof-of-concept first solution, we will test other updatable statistics (e.g., [25]) in place of ISOMER in the next version of PayLess.

4. QUERY OPTIMIZATION

PayLess's optimizer follows the typical bottom-up, cost-based, and dynamic programming approach [28]. That is, it first consid-



(a) Bushy tree (b) Left-deep tree

Figure 4: Bushy tree v.s. Left-deep tree

ers the best plan for single relations, then the best plan for joining two relations, and then for three relations, so on. On top of that, PayLess's optimizer considers bind joins $\overrightarrow{\bowtie}$as an access path in addition to the regular join $\bowtie$. The key feature of PayLess's optimizer is that it carries out *semantic query rewriting* to optimize its queries using the query results stored in the semantic store. Semantic query rewriting [23] is not new, but later we will explain why it is not included in limited access query optimizer (e.g., [17,27,45]) and why it is helpful to us here. We will also explain the limitations of current semantic query rewriting techniques in our setting and our solutions to unlock their potential and integrate them into our optimizer.

This section describes how to derive the optimal execution plan after parsing a SQL query. We first propose several techniques to reduce the plan search space and prove their correctness (see Section 4.1). After that, we illustrate the semantic query rewriting method used in PayLess (see Section 4.2). In the end, we end with some discussions about our query optimization approach (see Section 4.3).

4.1 Plan Space

When optimizing queries for limited access pattern data sources, *bushy trees* are included in the plan space to avoid plans with Cartesian products [27]. For example, consider a query that joins four relations U, R, S and T with access patterns: $U(x^f, y^f)$, $R(y^b, z^f)$, $S(t^f, w^f)$, $T(w^b, z^f)$. Since R has a bind attribute y, it must require values for attribute y to retrieve tuples. In the example, the only choice is thus to carry out a bind join $U \overrightarrow{\bowtie} R$. Similarly, since T has a bind attribute w, it must require values for attribute w to retrieve tuples. In the example, the only choice is thus to carry out a bind join $S \overrightarrow{\bowtie} T$. After that, the only way is to join them together by using a local join, resulting in a bushy tree like Figure 4a. So, if only left-deep plans are allowed, a "logical" cross product must be used to *logically connect* the relations like Figure 4b[4].

Including bushy trees would significantly enlarge the search space. In our problem setting, as our primary goal is to minimize the money-to-pay, we exclude bushy trees in our plan space because:

THEOREM 1. *Given any plan P, we can transform it to a left-deep plan P' such that $\phi(P) \geq \phi(P')$, where $\phi(\cdot)$ denotes the total price of a plan. In other words, the optimal plan must be one of the left-deep plans.*

PROOF. In what follows, we use the terms RESTful call, leaf node, and relation/table interchangeably.

First, we re-iterate a very important fact:

[4]The cross product is just logically connecting intermediate results U$\overrightarrow{\bowtie}$R and S$\overrightarrow{\bowtie}$T. Physically, (U$\overrightarrow{\bowtie}$R) joins (S$\overrightarrow{\bowtie}$T) is done by the DBMS, using any equi-join implementation like hash-join.

Fact *Only leaf nodes in P contribute to the price $\phi(P)$ because they represent RESTful calls to the data market. Therefore, $\phi(P)$ equals to the sum of prices of leaf nodes in P.*

Without loss of generality, we name the leaf nodes (RESTful calls) in P from left-to-right as: $C_1, C_2, \cdots, C_n$.

We write $P^{(k)}$ to denote that, for all leaf nodes of P, if named from left-to-right, the first k leaf nodes form a left-deep subtree. So, given a plan P with n leaf nodes, if we write $P^{(n)}$, we mean P is a complete left-deep tree. As an example, for the bushy tree P in Figure 4a. $P^{(1)}$ and $P^{(2)}$ hold. As another example, let P be the plan in Figure 4b, then we see that $P^{(1)}$, $P^{(2)}$, $P^{(3)}$ and $P^{(4)}$ all hold.

Now, we proceed to prove $\phi(P) = \phi(P^{(1)}) \geq \phi(P^{(2)}) \geq \cdots \geq \phi(P^{(n)})$. In the following, we first prove $\phi(P^{(1)}) = \phi(P)$ and then prove that for a given $1 \leq k \leq n-1$, we have $\phi(P^{(k+1)}) \leq \phi(P^{(k)})$.

<u>Base case: $k = 1$</u> $P^{(1)}$ simply means we just look at the left-most leaf nodes of P without moving any nodes, so the cost of the whole plan P is unchanged: $\phi(P^{(1)}) = \phi(P)$.

<u>General case: $\phi(P^{(k+1)}) \leq \phi(P^{(k)})$</u>

When C_{k+1} is C_k's uncle: Figure 5a illustrates this case. In this case, the left-most $k+1$ leaf nodes form a left-deep subtree. So, P^{k+1} holds. Note that we did not move any leaf node yet, so the plan cost would not change: $\phi(P^{(k+1)}) = \phi(P^{(k)})$.

When C_{k+1} is not C_k's uncle: Figure 5b illustrates this case. In this case, the uncle node of C_k, say U, must be a non-leaf node and its subtree contains C_{k+1}. Let T_F be the left-deep subtree rooted at F, the father of C_k. Further, we let G be the grandfather of C_k. Finally, we let T_{UL}, T_{UR} be the left and right subtrees rooted at U, respectively.

We now explain that making $P^{(k+1)}$ holds by joining T_F with C_{k+1} through a new node G' would not increase the overall plan cost. Figure 5c illustrates the resulting plan P' with $P'^{(k+1)}$ holds.

First, we see that the price of subtree T_F is the same among P and P'.

Second, the price of C_{k+1} is the same in both P and P' because C_{k+1} takes the same join result from T_F no matter G or G' is a bind join or a regular (local) join.

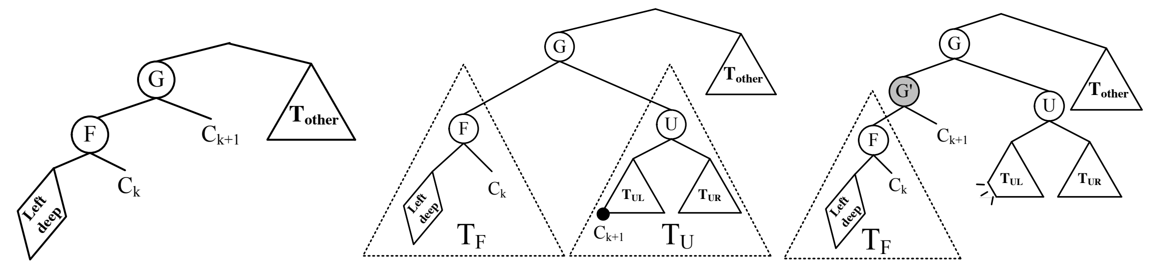
Now, we consider the price for each node (other than C_{k+1}) in T_{UL} and T_{UR} in P and P'. Let C_u be such a node. First, if C_u does not require any binding from C_{k+1}, then the price of C_u in P' is unchanged. Second, if C_u requires binding values from C_{k+1}, then the price of C_u depends on the number of distinct binding values from C_{k+1}. Note that in P', C_{k+1} has been joined with the others earlier than P, that causes the number of binding values to C_u possibly decreases. So, the price for C_u would not increase.

Finally, we look at the subtree T_{other}. As the result of the left operand of T_{other} remains the same, the price of T_{other} is unchanged.

As the price of any C_i in P would not increase, we have $\phi(P^{(k+1)}) \leq \phi(P^{(k)})$. $\square$

Traditional optimizers include only left-deep plans as a heuristic to improve the efficiency of the plan search. In PayLess, with Theorem 1, enumerating only left-deep plans is not a heuristic but with a guarantee that the optimal plan is not lost. Furthermore, in PayLess, including Cartesian product is not a problem because that would not contribute any extra data market transaction.

In addition to enumerating left-deep plans only (Theorem 1), PayLess's optimizer further trims the search space by first joining all relations that incur zero price to the data market. Those relations can either be local relations or relations whose required tuples can be found in the semantic store. In the following, we show that such



(a) When C_{k+1} is C_k's uncle (b) Before transformation P (c) After transformation P'

Figure 5: Illustration figures for Theorem 1

zero-price-relations-join-first idea retains the optimal plan in the plan space:

THEOREM 2. *Let $P = \langle C_1, C_2, \ldots, C_n \rangle$ be a left-deep plan with a leaf node (RESTful call) C_i whose price $\phi(C_i) = 0$. Then, the plan $P' = \langle C_i, C_1, \ldots, C_n \rangle$ has $\phi(P') \leq \phi(P)$.*

PROOF. We divide the other calls into two groups: (1) RESTful calls that executed before C_i, i.e. C_1 to C_{i-1}, and (2) RESTful calls that executed after C_i, i.e. C_{i+1} to C_n.

If we move C_i to the left-deepest node of P:

- $\phi(C_i)$ is unchanged and remains 0.
- $\phi(C_j)$ for $j > i$ is unchanged because the join results before executing C_j and the possible binding values for C_j are the same.
- $\phi(C_j)$ for $j < i$ cannot increase. If C_j does not use any binding attributes, then moving C_i before C_j would not increase $\phi(C_j)$. If C_j uses binding values from a bind join, then moving C_i before C_j would not increase (but may decrease) the number of bind join values for C_j, and that would not increase $\phi(C_j)$.

□

PayLess's optimizer applies Theorem 2 repeatedly and moves all zero price calls to the leftmost subtree of P. That way, the search space of PayLess's optimizer is further reduced.

Lastly, PayLess's optimizer would prune some candidate subplans during plan enumeration:

THEOREM 3. *When searching for the best plan for a set $\mathcal{C}$ of relations $C_1, C_2, \ldots, C_n$, if $\mathcal{C}$ can be partitioned into disjoint subsets $\mathcal{C}_1 \ldots \mathcal{C}_j$, where relations in $\mathcal{C}_i$ cannot join with relations in $\mathcal{C}_j$ (unless using Cartesian product $\times$). Then the best plan for $\mathcal{C}$ is $Best(\mathcal{C}_1) \times Best(\mathcal{C}_2) \times \ldots Best(\mathcal{C}_j)$, where $Best(\mathcal{C}_i)$ denotes the best plan for the set of relations in $\mathcal{C}_i$.*

PROOF. The proof is trivial because the relations in $\mathcal{C}_i$ cannot join with relations in $\mathcal{C}_j$, the price of calling $\mathcal{C}_j$ would not be influenced by $\mathcal{C}_i$. So, the best plan for $\mathcal{C}$ becomes simply connecting the best subplans of $\mathcal{C}_1 \ldots \mathcal{C}_j$ using Cartesian product. □

Consider a chain query that joins four relations: $\mathcal{C} = \{U(v,w), R(w,x), S(x,y), T(y,z)\}$. Assuming that the best plans determined for the pairs of relations are:

	$\{U,R\}$	$\{U,T\}$	$\{U,S\}$	$\{R,S\}$	$\{R,T\}$	$\{S,T\}$
Best Plan	$U \overrightarrow{\bowtie} R$	$U \times T$	$U \times S$	$R \overrightarrow{\bowtie} S$	$R \times T$	$S \overrightarrow{\bowtie} T$

So, when determining the best plan for 3-way join, the candidate plans that would be generated are:

	$\{U,R,S\}$	$\{U,R,T\}$	$\{U,S,T\}$	$\{R,S,T\}$
Candidate Plans	$(U \overrightarrow{\bowtie} R) \bowtie S$	$(U \overrightarrow{\bowtie} R) \bowtie T$	...	...
	$(U \overrightarrow{\bowtie} R) \overrightarrow{\bowtie} S$	$(U \times T) \bowtie R$	...	...
	$(U \times S) \overrightarrow{\bowtie} R$	$(U \times T) \overrightarrow{\bowtie} R$	...	...
	...	$(R \times T) \bowtie U$	...	...
	...	$(R \times T) \overrightarrow{\bowtie} U$	...	...

Observe that the set $\{U,R,T\}$ can be partitioned into two disjoint subsets: $\mathcal{C}_1 = \{U,R\}$ and $\mathcal{C}_2 = \{T\}$. So, we can apply Theorem 3 to determine the best plan for the set $\{U,R,T\}$ as $Best(U,R) \times T$, i.e., $(U \overrightarrow{\bowtie} R) \times T$. In other words, Theorem 3 eliminates many candidates (e.g., $(R \times T) \overrightarrow{\bowtie} U$) and eliminates their associated costing steps and semantic rewriting steps.

Let the total number of candidate plans in all levels of the dynamic programming approach be the size of the search space. For a chain query with n relations whose attributes are all free. The use of the above theorems can reduce the search space from $\approx 6^n - 5^n$ down to $\approx 2^{n'} + \frac{2}{3} \cdot n'^3$ with the optimal plan retained, where m is the number of zero price relations and $n' = n - m$. Specifically, the original plan space with dynamic programming is:

$$n + \sum_{k=2}^{n} \left(\binom{n}{k} \cdot \left(\sum_{i=1}^{k-1} \binom{k}{i} \cdot 4^{min\{i,k-i\}} \right) \right) \approx 6^n - 5^n$$

where k represents the level in dynamic programming (e.g., when $k = 2$, we consider joining two relations). At level k, there are $\binom{n}{k}$ size-k subsets to be examined. For each size k subset, we can form a plan by: (i) choosing a size i subset for the left subtree (and the complementary size $k - i$ subset for the right subtree), and (ii) deciding the binding attributes for the join (at root). For (ii), each call on the right subtree can bind with attributes from at most 2 calls from the left subtree; thus, there are 2·2=4 binding choices per call, and at most 4^{k-i} choices per plan. We can tighten this number to $4^{min\{i,k-i\}}$ when i is small and the left subtree can provide at most 4^i binding choices.

The plan space of PayLess's optimizer is:

$$4n' + \sum_{k=2}^{n'} \left(4 \cdot k \cdot (n' - k + 1) + \left(\binom{n'}{k} - (n' - k + 1) \right) \right)$$

$$\approx 2^{n'} + \frac{2}{3} \cdot n'^3$$

where m is the number of zero price relations and $n' = n - m$. Specifically by Theorem 2, we first build a plan with all local m relations. Then, in dynamic programming, we consider growing the plan by using the remaining $n' = n - m$ relations. At level k, there are $\binom{n'}{k}$ size-k subsets. We can divide them into (i) discon-

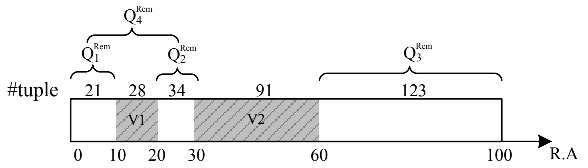


Figure 6: Generation of remainder queries

nected subsets (in which some relations must be joined by Cartesian product), and (ii) connected subsets. For the chain query, there are $n' - k + 1$ connected subsets and $\binom{n'}{k} - (n' - k + 1)$ disconnected subsets. For each disconnected subset, we can compute its best plan directly by Theorem 3. For each connected subset, we can obtain it by Theorem 1, i.e., combining a size-$(k-1)$ subset with a new call. There are k choices for the call and at most 2·2=4 binding choices for that call.

4.2 Semantic Query Rewriting

In PayLess, we store all RESTful queries issued to the data market and their corresponding results in the semantic store. The objective of doing so is to carry out *semantic query rewriting*, i.e., answer the queries using those stored results so as to reduce the amount of data retrieved from the data market. Semantic query rewriting falls into the category of rewriting queries using views [29, 48]. Given a query Q, a set $\mathcal{V}$ of RESTful queries and their corresponding stored results, the key step in semantic query rewriting is to compute the set $\mathcal{R}em(Q, \mathcal{V})$ of *remainder queries* [23]. The set $\mathcal{R}em(Q, \mathcal{V})$ essentially contains the set of RESTful queries that has to be sent to the data market in order to retrieve the tuples required by Q but not covered by $\mathcal{V}$.

Before we delve deeper, we first explain why optimizers for queries over remote data sources like [17, 27, 45] do not use semantic query rewriting. Consider our example query $Q1$ (page 1), which inquires about the daily temperature of Seattle in June 2014, has been issued, and its 30 resulting tuples (one tuple for each day in June) are stored in the semantic store. Assume that there is another query $Q2$ being issued, with $Q2$ shares the same query template like $Q1$ but the date ranges from May 2014 to July 2014 (3 months). Using semantic query rewriting, $Q2$ will generate two remainder queries: one asks for weather records in May (31 records; 1 transaction), another asks weather records in July (31 records; 1 transaction). The final result is then obtained by union the above with the stored results of $Q1$. The plan of using semantic query rewriting incurs a total of two calls to the external data source. In contrast, only one call to the external data source is required if $Q2$ is sent to the external data source without semantic query rewrite. So, in the context of minimizing the number of calls to external data sources, semantic query rewriting obviously is not a fruitful technique because it decomposes a call to several sub-calls.

Now, we show that how could we adapt semantic query rewriting to PayLess's optimizer to yield competitive plans for data market query processing. To illustrate, consider the example in Figure 6. The example assumes that the results of two queries V_1 and V_2 have been stored in the semantic store. Both V_1 and V_2 are range queries on an integer attribute A whose domain is $[0, 100]$. V_1 and V_2 respectively cover the ranges $[10, 20)$ and $[30, 60)$ on attribute A and have retrieved 28 and 91 tuples from table R. In what follows, we write a query Q in the form as

$$Q :-\ R_1(A[s, e], B = \beta, C), R_2(C, ..)$$

which means it joins R_1 and R_2 using C as the join attribute, and tuples in table R_1 have values in numeric attribute A fall between s and e and have values in categorical attribute B equal β.

Now, with V_1 and V_2, we assume the following query Q is posed:

$$Q :-\ R(A[0, 100])$$

Using the vanilla semantic query rewriting techniques, it will generate an invalid remainder query $Q^{Rem}_{invalid}$:

$$Q^{Rem}_{invalid} :-\ R(A[0, 10) \vee [20, 30) \vee [60, 100])$$

In data market, $Q^{Rem}_{invalid}$ is invalid because it involves disjunction, which is not supported by the access pattern of data market. Therefore, our first step to adapt semantic query rewriting techniques is to decompose remainder queries that violate the data source access patterns into a set of valid remainder (sub)queries. For the example above, PayLess will generate a set $\mathcal{R}em_1$ of remainder queries:

$Q_1^{Rem} :-\ R(A[0, 10))$ //21 tuples; 1 transaction
$Q_2^{Rem} :-\ R(A[20, 30))$ //34 tuples; 1 transaction
$Q_3^{Rem} :-\ R(A[60, 100])$ //123 tuples; 2 transactions

So, altogether, $\mathcal{R}em_1$ will cost a total 4 transactions.

Note that such straightforward decomposition may not yield the best plan. For example, the following is another possible set of remainder queries $\mathcal{R}em_2$:

$Q_4^{Rem} :-\ R(A[0, 30))$ //21+28+34= 83 tuples; 1 transaction
$Q_3^{Rem} :-\ R(A[60, 100])$//123 tuples; 2 transactions

The remainder query Q_4^{Rem}, although overlaps with stored query V_1, will still cost $\lceil (21 + 28 + 34)/100 \rceil = 1$ transaction. So, altogether, $\mathcal{R}em_2$ will cost a total 3 transactions only.

The example above illustrates a new and unique issue specific to the generation of remainder queries in data market. Specifically, we see that there are alternate ways to generate valid remainder queries and it is possible that a lower overall price can be achieved even when *a remainder query overlaps with a stored query*.

PayLess obviously does not want to miss the above opportunity when optimizing the queries. So, we have devised a remainder query generation method that leverages the above opportunity to reduce the overall price to access the data market.

We illustrate our idea using a more general example in Figure 7a. In the example, the query Q is a 2d-query that inquires table R:

$$Q :-\ R(A_1[30, 80], A_2[0, 50])$$

In the example, we assume there are ten RESTful queries $V_1, \ldots, V_{10}$ stored in the semantic store. Figure 7b shows the intersection of Q and the complement of $\mathcal{V}$, i.e., the data supposed to be retrieved from the data market. Denoting that space as $\overline{\mathcal{V}}$, there are alternate sets of remainder queries that can retrieve all the missing data. For example, consider the following set of remainder queries $\mathcal{R}em_3$:

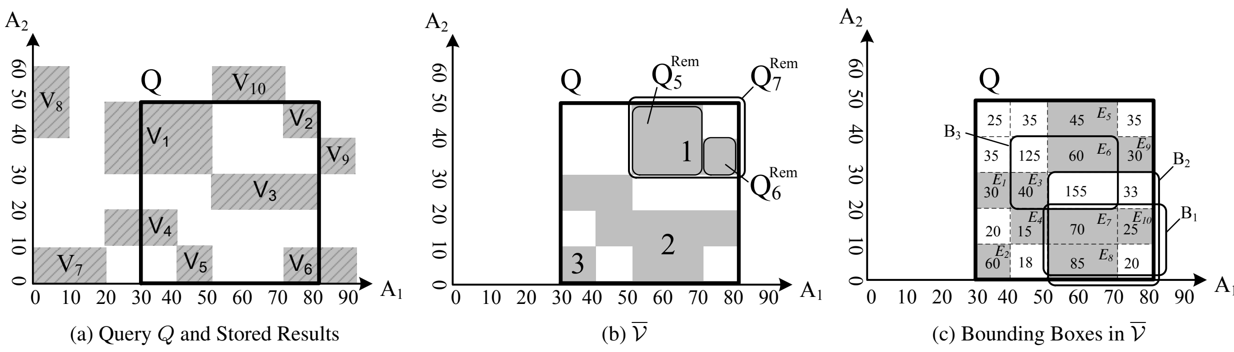
$Q_5^{Rem} :-\ R(A_1[50, 70), A_2[30, 50])$
$Q_6^{Rem} :-\ R(A_1[70, 80], A_2[30, 40])$

$\mathcal{R}em_3$ covers the missing data in region 1. Alternately, the following set of remainder queries $\mathcal{R}em_4$ can also cover data in region 1:

$Q_7^{Rem} :-\ R(A_1[50, 80), A_2[30, 50])$

From the above, we see that our goal boils down to finding a set of bounding boxes that cover all the data regions in $\overline{\mathcal{V}}$ using the least number of data market transactions.

To achieve a good solution, we use a two-step approach. The first step aims to generate a set of promising bounding box $\mathcal{B}$ candidates that cover different data regions in $\overline{\mathcal{V}}$. The bounding box candidates



(a) Query Q and Stored Results (b) $\overline{\mathcal{V}}$ (c) Bounding Boxes in $\overline{\mathcal{V}}$

Figure 7: Generation of remainder queries for data market

may possibly overlap with each other. The second step aims to extract from $\mathcal{B}$ the best set of bounding boxes that cover all the data regions in $\overline{\mathcal{V}}$ in minimum price.

We now elaborate the first step. Specifically, we begin with a decomposition of $\overline{\mathcal{V}}$ into a union $\mathcal{E}$ of disjoint *elementary boxes*. Figure 7c shows an example. On each dimension i, we collect a *separator* set S_i from the corners of each elementary box. For example, elementary box E_8 contributes values 50 and 70 to S_1 and contributes values 0 and 10 to S_2. Accounting for all elementary boxes, then we have $S_1 = \{30, 40, 50, 70, 80\}$ and $S_2 = \{0, 10, 20, 30, 40, 50\}$. Then, we exhaustively construct a set $\mathcal{B}$ of *bounding boxes*, where the extent of a bounding box $B \in \mathcal{B}$ on dimension i is picked from any two values in S_i. For example, the bounding box B_1 in Figure 7c has extent $[50, 80]$ on dimension A_1 and extent $[0, 20]$ on dimension A_2 when it picks values 50 and 80 from S_1 and values 0 and 20 from S_2. Each resulting bounding box represents a remainder query that covers certain data to be retrieved from the data market.

Algorithm 1 presents the pseudo-code of generating the bounding boxes, with powerful pruning rules to prune unpromising bounding boxes. First, it estimates the number of tuples falling into each elementary box in $\mathcal{E}$ from ISOMER (Lines 2–3). Figure 7c shows an illustration with those estimates. (We will discuss the case of insufficient/inaccurate statistics in the Section 4.3). Next, it enumerates a set of bounding boxes from the separator sets $S_1, S_2, ...S_d$, where d is the dimensionality of the query. It applies two pruning rules to discard unpromising bounding boxes.

The first pruning rule (Line 6) prunes a bounding box B if it is not tight. In other words, only *minimum bounding boxes* could stay. Consider the bounding boxes B_1 and B_2 in Figure 7c. They both contain the same set of elementary boxes E_7, E_8, E_{10} but B_2 contains B_1. Therefore, B_2 is not a minimum bounding box and is pruned. This makes sense because B_2 has to download an extra $155 + 33$ redundant tuples comparing with B_1.

The second pruning rule (Line 8) prunes a bounding box if its price is not smaller than the price sum of its individual elementary boxes. Consider bounding box B_3 Figure 7c. It requires $\lceil(125 + 60 + 40 + 155)/100\rceil = 4$ transactions. However, if E_3 and E_6 are individually retrieved, they collectively cost only $\lceil 40/100\rceil + \lceil 60/100\rceil = 2$ transactions. So, in this case, B_3 is not helpful and is pruned as well.

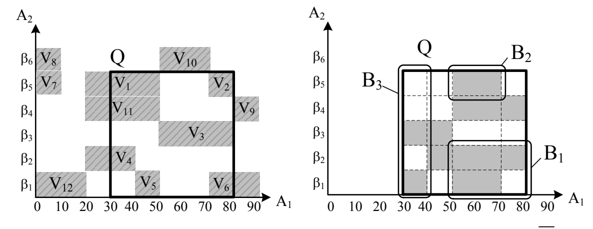
Algorithm 1 would enumerate $\binom{|S_i|}{2}^d$ bounding boxes for a d-dimensional query in the worst case. However, because of the high effectiveness of the pruning rules, the number of (minimum) bounding boxes considered is indeed much fewer than the worst case in practice.

The second step of our idea is to find the best subset of minimum bounding boxes (generated from Algorithm 1) that cover all the elementary boxes (all missing data) in minimum price. This is a weighted set cover problem [22]. Specifically, the weighted set cover problem states that, given (1) a set of elements $\mathcal{E} = \{E_1, E_2, ...\}$ and (2) a family $\mathcal{B}$ of subsets of $\mathcal{E}$, in which each subset in $\mathcal{B}$ is associated with a $cost_i$, find a collection of subsets, namely the cover, $Cover \subseteq \mathcal{B}$, whose union of the elements in $Cover$ is $\mathcal{E}$ and the sum of cost of elements in $Cover$ is the minimum. In our context, we have (1) $\mathcal{E}$ as all elementary boxes and (2) $\mathcal{B}$ as the set of candidate minimum bounding boxes returned by Algorithm 1, $cost_i$ is referred as a bounding box 's estimated transactions. To solve this $\mathcal{NP}$-hard problem, we use the greedy algorithm in [22] that runs in $O(|\mathcal{B}| \cdot |\mathcal{E}|)$ time with $(1 + \ln(|\mathcal{B}|))$ approximation ratio.

The generation of bounding boxes for queries with *categorical attributes* is illustrated as follows. Figure 8a shows an example similar to the previous one but with attribute A_2 now becomes a categorical attribute with the following domain: $\{\beta_1, \beta_2, \beta_3, \beta_4, \beta_5, \beta_6\}$. We remark that there are no stored queries that can span across multiple categorical values because of the limitation of the access interface.

Algorithm 1 Generating Candidate Remainder Queries

```
    Input (elementary boxes $\mathcal{E}$, separator sets $\{S_1, S_2, ..., S_n\}$)
    Output (A collection of minimum bounding boxes $\mathcal{B}$)
 1: initialize $\mathcal{B}$
 2: for each elementary box $E_i$ in $\mathcal{E}$ do
 3:     $E_i$.price ← estimate the price of $E_i$
 4: enumerate every possible bounding box $B$ using the separator
    sets $S_1, S_2, \ldots, S_n$.
 5: for each bounding box $B$ do
 6:     if $B$ is a minimum bounding box then        ▷ pruning rule 1
 7:         estimate the price of $B$
 8:         if $B$.price < $\sum_{E_i \in B} E_i$.price then    ▷ pruning rule 2
 9:             insert $B$ into $\mathcal{B}$
10: return $\mathcal{B}$
```



(a) Query Q and Stored Results (b) Bounding Boxes in $\overline{\mathcal{V}}$

Figure 8: Generation of remainder queries with a categorical attribute A_2

Figure 8b shows the corresponding space $\overline{\mathcal{V}}$. Since A_2 is a categorical attribute, the bounding box B_1, which represents the following remainder query, is invalid:

$$:-\ \ R(A_1[50,80), A_2 = \beta_1 \vee A_2 = \beta_2)$$

Therefore, we will only generate bounding boxes that span either one value or the whole domain of a categorical attribute. For example, bounding boxes B_2, which represents the following remainder query, is valid and would be generated:

$$:-\ \ R(A_1[50,70), A_2 = \beta_5)$$

Similarly, bounding boxes B_3, which represents the following remainder query, is also valid and would be generated:

$$:-\ \ R(A_1[30,40))$$

The generation of bounding boxes for queries with *bind joins* is illustrated as follows. Consider a relation U with binding pattern $U(A_1^f, A_2^f)$ and a relation S with binding pattern $S(A_2^b, A_3^f)$, where all attributes are integer attributes. Further, consider a query V that joins U and S:

$$V :-\ \ U(A_1[2,3], A_2), S(A_2, A_3[10,15])$$

V needs a bind join because A_2 is a bind attribute. So, assume that there are four tuples t_1, t_2, t_3, and t_4 in U having values within the range $[2,3]$ in attribute A_1 and their corresponding values in attribute A_2 are 2, 5, 9, and 10, respectively. Then, the bind join is carried out with S by binding the values 2, 5, 9, and 10 to S's attribute A_2. Note that when retrieving tuples from S whose attribute A_2 has a value, say, 2, those tuples have to satisfy the other condition $A_3[10,15]$ as well. Figure 9a illustrates the above process.

Now, assume the query results of V are stored in the semantic store and let us consider a query Q that shares the same query template as V but with a different query range:

$$Q :-\ \ U(A_1[2,5], A_2), S(A_2, A_3[8,18])$$

Note that in this case, assuming that we can estimate that two tuples t_x and t_y will be retrieved from U for $A_1 = 4$, one tuple t_z will be retrieved from U for $A_1 = 5$ (we don't need to estimate the cardinality for $A_1 = 2$ and $A_1 = 3$ because we know the exact cardinality from V), exact values of t_x, t_y, t_z's attribute A_2 are still unknown (denoted as ? in Figure 9b). In this case, it will generate $\overline{\mathcal{V}}$ like Figure 9c. Consequently, when enumerating the set of candidate bounding boxes, we can generate a bounding box for each individual elementary box (e.g., B_1), for a range of known values (e.g., B_2), or for the whole domain (e.g., B_3). In contrast, we cannot generate a bounding box like B_4 because the exact value for A_2 of t_z is actually unknown.

Algorithm 2 shows the pseudo code of PayLess optimization. It is self-explanatory and mainly summarizes what we have discussed above, so we do not give it a walkthrough here.

4.3 Discussion

We end this section with a number of discussions about our query optimization approach. First, as in traditional cost-based query optimization, our approach relies on metadata like histograms. In the beginning when no rich statistics such as value distributions are available, PayLess's optimizer would carry out the cardinality estimation using the basic textbook methods (e.g., using the domain size and uniform distribution assumption).

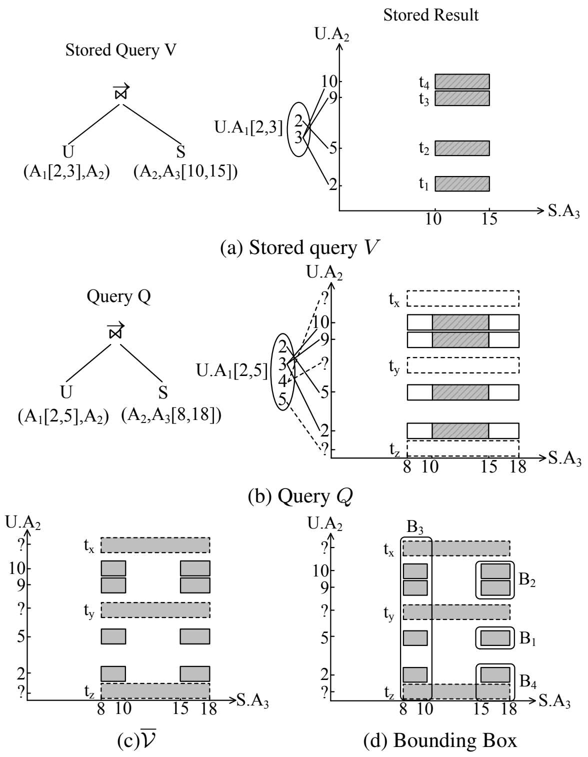


Figure 9: Example for 2D-Bind

Algorithm 2 PayLess Query Optimization

```
    Input ( a query $Q$, a set $\mathcal{V}$ of RESTful queries and their stored
    results, the metadata $\mathcal{M}$ for cost estimation )
    Output ( the optimal plan $P^*$ : $Best(Q)$ for the query $Q$ )
 1: $\mathcal{R}_{local} \leftarrow \{C_i \in Q : \phi(C_i) = 0\}$; $\mathcal{R}' \leftarrow \{C_i \in Q\} - \mathcal{R}_{local}$
 2: $P_{local} \leftarrow$ the best subplan for $\mathcal{R}_{local}$; found by offloading to a
    DBMS's optimizer
 3: for each $C_i \in Q$ do                                  ▷ size-1 subplans
 4:     $Best(C_i) \leftarrow$ SemanticRewrite($C_i, \mathcal{V}, \mathcal{M}$)
 5: execute Line 1 again to update $\mathcal{R}_{local}$ and $\mathcal{R}'$
 6: for each $k$ from 2 to $|\mathcal{R}'|$ do                           ▷ Theorem 2
 7:     for each size-$k$ subset $\mathcal{R}^k$ of $\mathcal{R}'$ do
 8:         if $\mathcal{R}_{local} \cup \mathcal{R}^k$ form $\ell$ disjoint subsets then          ▷
    Theorem 3
 9:             $Best(\mathcal{R}^k) \leftarrow Best(\mathcal{R}_1^k) \times Best(\mathcal{R}_2^k) \times \cdots \times$
    $Best(\mathcal{R}_\ell^k)$
10:         else for each call $C_i \in \mathcal{R}^k$                     ▷ Theorem 1
11:             rewrite $C_i$ as $\overrightarrow{C_i}$ by using binding from $P_{local} \bowtie$
    $Best(\mathcal{R}^k - C_i)$
12:             $P_{bind} \leftarrow$ SemanticRewrite($\overrightarrow{C_i}, \mathcal{V}, \mathcal{M}$)
13:             $P_{temp} \leftarrow Best(\mathcal{R}^k - C_i) \bowtie Best(C_i)$
14:             if $\phi(P_{bind}) \leq \phi(Best(C_i))$ then
15:                 $P_{temp} \leftarrow Best(\mathcal{R}^k - C_i) \overrightarrow{\bowtie} P_{bind}$
16:             update $Best(\mathcal{R}^k) \leftarrow P_{temp}$ if $\phi(Best(\mathcal{R}^k)) \geq$
    $\phi(P_{temp})$
```

Second, answering a query using the stored query results may include obsolete tuples if datasets permit *in-place* data update. However, so far the datasets we found in Windows Azure Marketplace are append-only. In case in-place data update exists, we will introduce several *consistency levels* into PayLess. That would allow organizations that install PayLess to choose between consistency levels like *(i) weak consistency*, *(ii) X-week consistency*, or *(iii) full consistency*. Weak consistency means all RESTful queries and their results are stored in the semantic store (with obsolete results get updated if new results are retrieved). Under weak consistency, semantic query rewriting is always enabled. Queries may however return partially obsolete results when there are in-place updates in the data market's datasets because it reuses some obsolete stored results. Strong consistency means semantic query writing is simply disabled and PayLess always go to the data market to obtain the latest results. X-week consistency is in the middle, it enables semantic query rewriting using query results retrieved from the past X weeks. The three options are trade-off between price-to-pay and the freshness of the result.

5. EVALUATION

PayLess aims to help organizations to pay less when their end users have to query against the data market. Without PayLess, one option is to employ query optimizers for data sources with limited access pattern because those optimizers at least consider binding patterns and bind joins in their architecture. Another option is to download all required tables from the data market upfront and carry out local processing afterwards. Notice this "Download All" option is not always bad. First, it is optimal if the queries have to scan the whole dataset. In this case, once the whole dataset is downloaded, all queries can work on the downloaded data locally. Second, if the number of transactions incurred by user queries would eventually exceed the number of transactions required to download the complete data set, then downloading the whole dataset upfront would be a more economical option. However, we re-iterate that it is always tough to predict how many user queries would eventually be issued in practice. Consider that the users walk away from the dataset forever after issuing just a few queries (maybe due to no interesting information is found), then downloading the whole dataset would become a very costly option.

In this section, we evaluate the effectiveness of PayLess using both real data and synthetic data. Specifically, we extract query templates from a meteorological application that involves queries to the Worldwide Historical Weather (WHW) [13] and Environmental Hazard Rank (EHR) [6] datasets in Windows Azure Marketplace. Table 1 lists the query templates and Figure 1a lists the sizes of the tables. We generate *valid* query instances from those templates by randomly assigning values to the parameters. A query instance is valid if it returns non-empty results (e.g., we would not instantiate Q_4 with a country equals 'USA' but a zip code in Germany). We also use the TPC-H workload in the experiments. We generate 1G of TPC-H data and 1G of TPC-H skew data [19] with $zipf = 1$. All parametric attributes in TPC-H queries are set as free attributes in the experiments. We set the relations `Nation` and `Region` local. By default, we set 100 tuples as one transaction (i.e., $t = 100$).

Overall effectiveness. We first study the overall effectiveness of PayLess under different workloads and datasets. For comparison, we include the results of using [27] to optimize the queries (denoted as "Minimizing Calls" in the figure). We also include the results of disabling semantic query rewriting (SQR) in PayLess (denoted as "PayLess w/o SQR" in the figure). We respectively generated q query instances per template. The query instances are issued in a random order and the results are reported as an average over 30 repeated experiments. In this experiment, we set $q = 10$ and $q = 200$ for TPC-H workload and real workload, respectively.

Figure 10a illustrates the total (cumulative) number of data market transactions used to answer the real queries. Except the "Download All" option, when more queries are issued, the total (cumulative) number of data market transactions increases. Comparing with those data buyers who recklessly download the whole dataset upfront, PayLess can now help them to answer the queries using about two orders of transactions fewer. The number of transactions used by PayLess grows slowly because many queries are rewritten using the stored results in the semantic cache. PayLess can answer the queries using about an order of transactions fewer than queries optimized using [27]. That is because semantic query rewriting (SQR) is not applicable to their setting but is a powerful helper here in our data market setting. When we disable SQR, PayLess still outperforms [27]. That is because PayLess can find optimal plans in a reduced search space using progressively refined statistics. In contrast, [27] has to find plans in a larger search space (including bushy trees) using heuristics.

Figures 10b and c show the results of using TPC-H workload. TPC-H queries scan a large portion of data. Therefore, without rewriting the queries using the stored data, each query optimized by [27] and PayLess (if SQR is disabled) would retrieve a large portion of the data from the data market, and those data are largely overlapping with each other. That explains why they are worse than "Download All", because the latter only downloads the whole dataset once. When PayLess is in full power with semantic query rewriting, we see that the subsequent queries can largely reuse the stored results, thereby saving a lot more transactions than "Download All" until about 80 queries have been issued. When about 80 queries have been issued, all the data required by TPC-H queries (indeed the whole TPC-H dataset) are stored by PayLess, therefore PayLess would not repeatedly retrieve the data from the data market anymore. From the above experimental results, we regard PayLess to be practically better than "Download All" in all means because nobody could have known the number of queries to be issued and the distribution of the data in practice. A data buyer can freely query against any dataset in the data market and walk away from that dataset anytime — she does not need to worry whether it is worth or not to download the whole dataset in the beginning, or switch to download the whole dataset when she finds out that she has to ask more queries after she has burned a certain amount of money.

Influence of number of tuples per transaction. We next study whether the effectiveness of PayLess would be influenced by the number of tuples per transaction, which could be a different value in different data markets. Since [27] is consistently outperformed by PayLess in all our experiments, so we remove it, together with PayLess with semantic query rewriting disabled, from our discussion.

Figure 11 shows the effectiveness of PayLess when we vary the number of tuples per transaction t. Note that when t is smaller, more transactions are required to retrieve the same number of tuples from the data market. Therefore, the number of transactions used by both PayLess and "Download All" must increase. Nevertheless, we see that the effectiveness of PayLess is not influenced by that data market parameter. PayLess still outperforms "Download All" under real data in all cases. In addition, it still outperforms "Download All" on TPC-H and TPC-H skew data until the whole dataset is retrieved.

Influence of number of query instances per query template. We next study whether the effectiveness of PayLess would be influenced by q, the number of query instances per query template. Figure 12 shows that the effectiveness of PayLess is not influenced by that parameter. We see that PayLess still consistently outperforms "Download All" on real data in all cases. In addition, it still outper-

Table 1: Query Templates on Real Data Sets

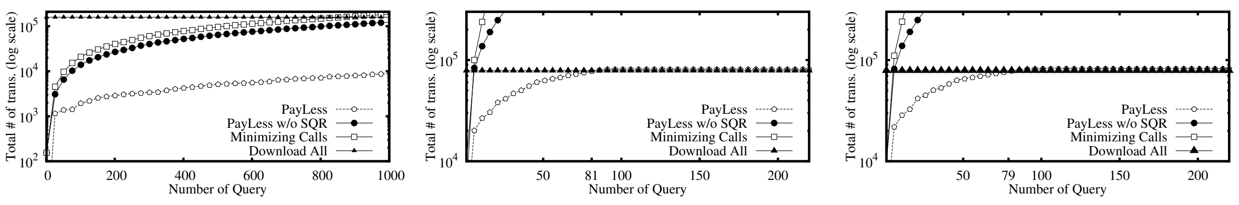
Q_1	`SELECT * FROM Weather`
	`WHERE Weather.Country = ?  AND Weather.Date >= ?  AND Weather.Date <= ?`

Q_2	`SELECT COUNT(ZipCode) FROM Pollution`
	`WHERE Pollution.Rank >= ?  AND Pollution.Rank <= ?`

Q_3	`SELECT AVG(Temperature) FROM Station, Weather`
	`WHERE Station.Country = Weather.Country = ?  AND Weather.Date >= ?  AND`
	`      Weather.Date <= ?  AND Station.StationID = Weather.StationID`
	`GROUP BY City`

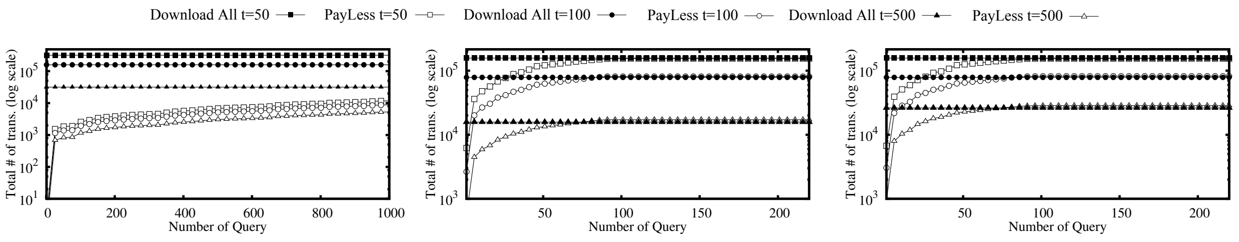
Q_4	`SELECT Temperature FROM Station, Weather, ZipMap`
	`WHERE Station.Country = Weather.Country = ?  AND ZipMap.ZipCode = ?  AND`
	`      Weather.Date >= ?  AND Weather.Date <= ?  AND`
	`      Station.StationID = Weather.StationID AND Station.City = ZipMap.City`

Q_5	`SELECT * FROM Pollution, Station, Weather, ZipMap`
	`WHERE Station.Country = Weather.Country = ?  AND Weather.Date >= ?  AND`
	`      Weather.Date <= ?  AND Pollution.Rank >= ?  AND Pollution.Rank <= ?  AND`
	`      Pollution.ZipCode = ZipMap.ZipCode AND ZipMap.City = Station.City AND`
	`      Station.StationID = Weather.StationID`



(a) Real data (b) TPC-H (c) TPC-H skew

Figure 10: Overall Effectiveness



(a) Real data (b) TPC-H (c) TPC-H skew

Figure 11: Varying the number of results t per transaction

forms “Download All" on TPC-H and TPC-H skew data until the whole dataset is retrieved.

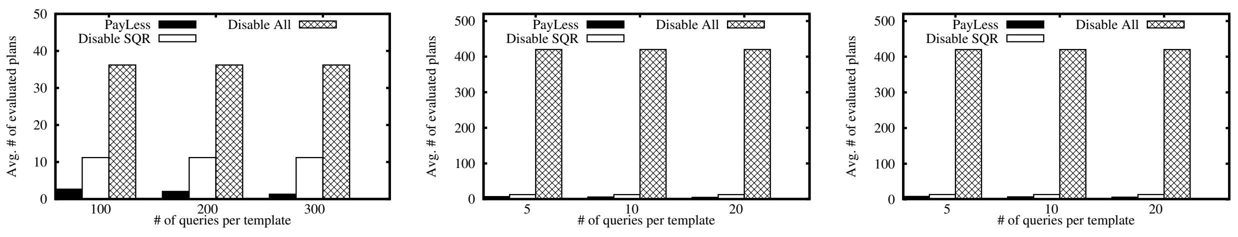
Influence of data size. We also study whether the effectiveness of PayLess would be influenced when the size of the data is varied. As we cannot control the size of the real data, we control only the size of the synthetic data.

Note that when the data size increases, “Download all” needs more transactions to download the whole dataset. But PayLess also needs to retrieve more tuples for each query. Figure 13 shows that PayLess still outperforms “Download All" on TPC-H and TPC-H skew data until the whole dataset is retrieved.

Effectiveness of search space reduction techniques. We have also carried out an experiment to evaluate the effectiveness of our techniques devoted to reducing the search space size. Figure 14 shows the average number of candidate (sub)plans for all query instances under our default setting. We report the case when (i) SQR is disabled (Disable SQR), (ii) both SQR and search space pruning (Theorems 1 to 3) are disabled (Disable All), and (iii) nothing is disabled (PayLess). We can see that our techniques significantly reduce the search space by orders of magnitude. This is actually what enables us to look for optimal plans. We notice that enabling SQR indeed reduces the search plan because SQR would cause some relations become local, which can then trigger Theorem 2. This also explains why the average number of candidate (sub)plans PayLess has to considered decreases when we increase the number of query instances generated for each template. That is because if we increase the number of query instances generated for each template, that would retrieve more data from the data market, which in turn increases the chance of using Theorem 2 to reduce the search space.

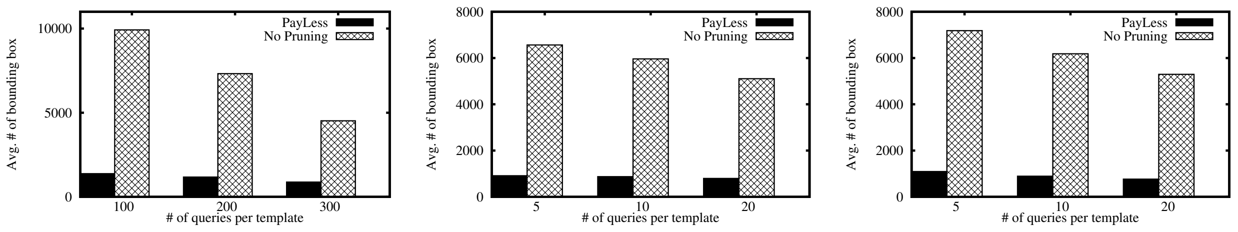
Effectiveness of bounding box pruning. Our last experiment is to evaluate the effectiveness of the bounding box pruning rules in Algorithm 1. Figure 15 shows the average number of bounding boxes generated for all query instances under our default setting. We see that the two pruning rules can reduce about an order bounding boxes generated.

Efficiency. In all experiments, the execution time of a query is, as usual, dominated by the RESTful calls to the data seller. Nevertheless, a query can still finish within seconds. The query optimization and the query execution part done by PayLess on the data buyer side all finish within milliseconds. We omit the detailed numbers here for space reasons.



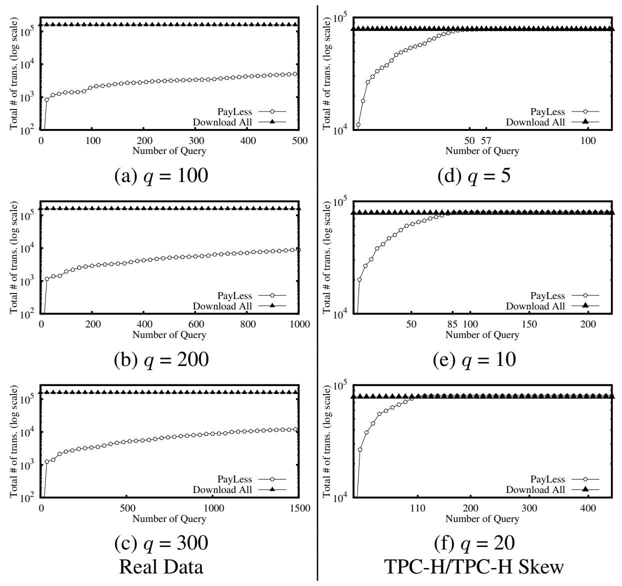
(a) Real data (b) TPC-H data (c) TPC-H skew data

Figure 14: Effectiveness of search space reduction techniques



(a) Real data (b) TPC-H data (c) TPC-H skew data

Figure 15: Effectiveness of bounding box pruning rules



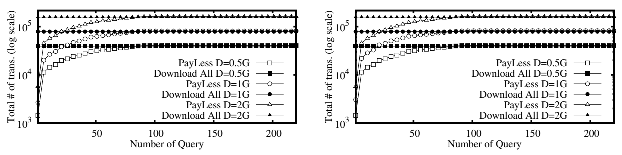
(a) q = 100 (d) q = 5

(b) q = 200 (e) q = 10

(c) q = 300 (f) q = 20

Real Data TPC-H/TPC-H Skew

Figure 12: Varying the number of query instances (q) per template



(a) TPC-H data (b) TPC-H skew data

Figure 13: Varying data size

6. RELATED WORK

To the best of our knowledge, this paper is the first to tackle the issue of optimizing queries that access the data market. So far, projects related to the data market are mainly developed for *query market*. In their setting, the query market can support SQL. A data buyer sends a SQL query that accesses a dataset in the query market. The query market computes the results of the query and returns the answer to the buyer. The research focus is how to set the price of arbitrary SQL queries (e.g. [15,16,30,31,37,39,47]). The setting of query market is different from our data market setting. Specifically, existing data market like Windows Azure Marketplace [1] and Xignite [14] are still charging data buyers according to the size of retrieved data.

In terms of problem setting, PayLess is indeed more similar to projects that support queries over remote data sources with limited access patterns (e.g., [17,20,24,27,33–36,40,45]). Nevertheless, as mentioned, all these projects have a very different focus with us — they are designed to minimize the number of calls to external data and/or the execution time. In contrast, PayLess focuses on minimizing the amount of intermediate retrieved data measured in terms of data market transactions. Besides, the optimization of distributed queries with semi-join/magic sets [18,43] are similar to PayLess; however, they do not consider limited access patterns.

In terms of implementation, PayLess has borrowed the idea of learning optimizer from LEO [46] and has used feedback driven histogram ISOMER [44]. However, PayLess has to develop its own architecture, construct its own plan search space, and devise its own semantic query rewriting technique (e.g., [21,23,32,41]) to fit the data market. In computational geometry, the problem of partitioning an orthogonal polygon into rectangles (PiR) [26] is similar to our remainder query generation problem, but they are not the same. Using Figure 7b as an example, the PiR problem would NOT consider Q_7^{Rem}, which contains some empty regions. In contrast, in our context, Q_7^{Rem} could be a good choice according to our cost function.

7. CONCLUSION

This paper presents PayLess, a system that helps data buyers to freely query against any dataset in the data market and walk away from that dataset anytime. The data buyers do not need to worry whether it is worth or not to download the whole dataset in the beginning. They can simply issue their queries to PayLess and PayLess optimizes their queries with the objective of minimizing their money-to-pay-to-data-sellers. Currently, our use-case does not cover many end users using PayLess simultaneously. When it does, we will incorporate multi-query optimization in PayLess if users are willing to defer theirs to become a batch.

Acknowledgments

The research is partly supported by grant GRF 152201/14E from Hong Kong RGC. The research is partly supported by the Research Grants Council of Hong Kong (GRF PolyU 520413 and 521012) and a research gift from Microsoft Hong Kong.

8. REFERENCES

[1] Azure data marketplace. `https://datamarket.azure.com`.

[2] Data markets compared. `http://radar.oreilly.com/2012/03/data-markets-survey.html`.

[3] data.gov. `https://www.data.gov/`.

[4] Datamarket. `https://datamarket.com/`.

[5] Discussing data markets in new york city. `http://cloudofdata.com/2013/02/discussing-data-markets-in-new-york-city/`.

[6] Environmental hazard rank data. `http://datamarket.azure.com/dataset/edr/environmentalhazardrank`.

[7] Esri. `http://www.esri.com/`.

[8] Factual. `http://www.factual.com/`.

[9] Infochimps. `http://www.infochimps.com`.

[10] Is infochimps running from the data market business? `http://cloudofdata.com/2013/02/is-infochimps-running-from-the-data-market-business/`.

[11] Wolfram alpha. `http://www.wolframalpha.com/`.

[12] World bank. `http://www.worldbank.org/`.

[13] Worldwide historical weather data. `https://datamarket.azure.com//dataset//weathertrends//worldwidehistoricalweatherdata`.

[14] Xignite data market. `http://www.xignite.com`.

[15] S. Al-Kiswany, H. Hacigümüs, Z. Liu, and J. Sankaranarayanan. Cost exploration of data sharings in the cloud. In *EDBT*, 2013.

[16] M. Balazinska, B. Howe, and D. Suciu. Data markets in the cloud: An opportunity for the database community. *PVLDB*, 2011.

[17] A. Calì and D. Martinenghi. Querying data under access limitations. In *ICDE*, 2008.

[18] S. Chaudhuri. An overview of query optimization in relational systems. In *PODS*, 1998.

[19] S. Chaudhuri and V. Narasayya. Program for tpc-d data generation with skew. 2012.

[20] S. Chaudhuri and K. Shim. Query optimization in the presence of foreign functions. In *VLDB*, 1993.

[21] B. Chidlovskii and U. M. Borghoff. Semantic caching of web queries. *VLDB J.*, 2000.

[22] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):pp. 233–235, 1979.

[23] S. Dar, M. J. Franklin, B. Þ. Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *VLDB*, 1996.

[24] O. M. Duschka and A. Y. Levy. Recursive plans for information gathering. In *IJCAI (1)*, 1997.

[25] A. El-Helw, I. F. Ilyas, W. Lau, V. Markl, and C. Zuzarte. Collecting and maintaining just-in-time statistics. In *ICDE*, 2007.

[26] D. Eppstein. Graph-theoretic solutions to computational geometry problems. *CoRR*, 2009.

[27] D. Florescu, A. Y. Levy, I. Manolescu, and D. Suciu. Query optimization in the presence of limited access patterns. In *SIGMOD*, 1999.

[28] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008.

[29] A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 2001.

[30] P. Koutris, P. Upadhyaya, M. Balazinska, B. Howe, and D. Suciu. Query-based data pricing. In *PODS*, 2012.

[31] P. Koutris, P. Upadhyaya, M. Balazinska, B. Howe, and D. Suciu. Toward practical query pricing with querymarket. In *SIGMOD*, 2013.

[32] D. Lee and W. W. Chu. Towards intelligent semantic caching for web sources. *J. Intell. Inf. Syst.*, 2001.

[33] C. Li. Computing complete answers to queries in the presence of limited access patterns. *VLDB J.*, 2003.

[34] C. Li and E. Y. Chang. Query planning with limited source capabilities. In *ICDE*, 2000.

[35] C. Li and E. Y. Chang. Answering queries with useful bindings. *ACM Trans. Database Syst.*, 2001.

[36] C. Li and E. Y. Chang. On answering queries in the presence of limited access patterns. In *ICDT*, 2001.

[37] Z. Liu and H. Hacigümüs. Online optimization and fair costing for dynamic data sharing in a cloud data market. In *SIGMOD*, 2014.

[38] V. Markl, V. Raman, D. E. Simmen, G. M. Lohman, and H. Pirahesh. Robust query processing through progressive optimization. In *SIGMOD*, 2004.

[39] A. Muschalle, F. Stahl, A. Löser, and G. Vossen. Pricing approaches for data markets. In *BIRTE*, 2012.

[40] A. Rajaraman, Y. Sagiv, and J. D. Ullman. Answering queries using templates with binding patterns. In *PODS*, 1995.

[41] Q. Ren, M. H. Dunham, and V. Kumar. Semantic caching and query processing. *IEEE Trans. Knowl. Data Eng.*, 2003.

[42] F. Schomm, F. Stahl, and G. Vossen. Marketplaces for data: an initial survey. *SIGMOD Record*, 2013.

[43] P. Seshadri, J. M. Hellerstein, H. Pirahesh, T. Y. C. Leung, R. Ramakrishnan, D. Srivastava, P. J. Stuckey, and S. Sudarshan. Cost-based optimization for magic: Algebra and implementation. In *SIGMOD*, 1996.

[44] U. Srivastava, P. J. Haas, V. Markl, M. Kutsch, and T. M. Tran. Isomer: Consistent histogram construction using query feedback. In *ICDE*, 2006.

[45] U. Srivastava, K. Munagala, J. Widom, and R. Motwani. Query optimization over web services. In *VLDB*, 2006.

[46] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. Leo - db2's learning optimizer. In *VLDB*, 2001.

[47] P. Upadhyaya, M. Balazinska, and D. Suciu. How to price shared optimizations in the cloud. *PVLDB*, 2012.

[48] H. Z. Yang and P.-Å. Larson. Query transformation for psj-queries. In *VLDB*, 1987.

Learning to Rank Adaptively for Scalable Information Extraction

Pablo Barrio
Columbia University
pjbarrio@cs.columbia.edu

Gonçalo Simões
INESC-ID and IST, Universidade de Lisboa
goncalo.simoes@tecnico.ulisboa.pt

Helena Galhardas
INESC-ID and IST, Universidade de Lisboa
helena.galhardas@tecnico.ulisboa.pt

Luis Gravano
Columbia University
gravano@cs.columbia.edu

ABSTRACT

Information extraction systems extract structured data from natural language text, to support richer querying and analysis of the data than would be possible over the unstructured text. Unfortunately, information extraction is a computationally expensive task, so exhaustively processing all documents of a large collection might be prohibitive. Such exhaustive processing is generally unnecessary, though, because many times only a small set of documents in a collection is useful for a given information extraction task. Therefore, by identifying these useful documents, and not processing the rest, we could substantially improve the efficiency and scalability of an extraction task. Existing approaches for identifying such documents often miss useful documents and also lead to the processing of useless documents unnecessarily, which in turn negatively impacts the quality and efficiency of the extraction process. To address these limitations of the state-of-the-art techniques, we propose a principled, learning-based approach for ranking documents according to their potential usefulness for an extraction task. Our low-overhead, online learning-to-rank methods exploit the information collected during extraction, as we process new documents and the fine-grained characteristics of the useful documents are revealed. Then, these methods decide when the ranking model should be updated, hence significantly improving the document ranking quality over time. Our experiments show that our approach achieves higher accuracy than the state-of-the-art alternatives. Importantly, our approach is lightweight and efficient, and hence is a substantial step towards scalable information extraction.

1. INTRODUCTION

Information extraction systems are complex software tools that discover structured information in natural language text. For instance, an information extraction system trained to extract tuples for an *Occurs-in*(*NaturalDisaster*, *Location*) relation may extract the tuple <tsunami, Hawaii> from the sentence: "*A tsunami swept the coast of Hawaii.*" Having information in structured form enables more sophisticated querying and data mining than what is possible over the natural language text. Unfortunately, information extraction is a time-consuming task. A state-of-the-art information extraction system to extract *Occurs-in* tuples may take more than two months to process a collection containing about 1 million documents. Since document collections routinely contain several millions of documents, improving the efficiency and scalability of the extraction process is critical, even over highly parallel computation environments.

Interestingly, extracting a relation of interest with a properly trained information extraction system rarely requires processing all documents of a collection: Many times only a small set of documents produces tuples for a given relation, because relations tend to be topic-specific, in that they are associated mainly with documents about certain topics. For example, only 1.69% out of the 1.03 million documents in collections 1-5 from the TREC conference[1] produce *Occurs-in* tuples when processed with a state-of-the-art information extraction system and, not surprisingly, most of these documents are on environment-related topics. If we could identify the small fraction of documents that lead to the extraction of tuples, we would extract all tuples while decreasing the extraction time by over 90% without any need to change the information extraction system.

To identify the documents that produce tuples for an extraction task, which we refer to as the *useful* documents, existing techniques (e.g., QXtract [2], PRDualRank [14], and FactCrawl [7]) are based on the observation that such documents tend to share words and phrases that are specific to the extraction task at hand. For example, documents containing mentions of earthquakes—hence useful for the *Occurs-in* relation—many times include words like "richter" or "hypocenter." These words and phrases can then be used as keyword queries, to retrieve from the collection the (hopefully useful) documents that the extraction system will then process. To discover these words and phrases, a critical step in the process, these techniques analyze a sample of documents from the collection of interest. The size of this document sample is necessarily small to keep the overhead of the querying approach at reasonable levels.

Unfortunately, small document samples are unlikely to reflect the typically large variations in language and content

[1]http://trec.nist.gov/data.html

10.5441/002/edbt.2015.22

that useful documents for an extraction task may exhibit. For example, a document sample for *Occurs-in* may not include any documents on (relatively rare) volcano eruptions, and hence these techniques may fail to derive queries such as [lava] or ["sulfuric acid"] that would retrieve relevant, volcano-related documents. As a result, the queries from existing techniques may suffer from low recall during extraction. Furthermore, precision is also compromised: standard keyword search identifies documents whose topic is relevant to the queries, without considering their relevance to the information extraction task at hand.

To alleviate the precision-related issue above, FactCrawl [7] moves a step beyond keyword search: after retrieving documents with sample-derived keyword queries, FactCrawl reranks the documents according to a simple function of the number and "quality"—based on their F-measure [27]—of the queries that retrieved them, thus helping prioritize the extraction effort. However, FactCrawl exhibits two key weaknesses: (i) for document retrieval and ranking, FactCrawl relies on queries derived, once and for all, from a document sample, and hence suffers from the sample-related problems discussed above; (ii) for document ranking, FactCrawl relies on a coarse, query-based document scoring approach that is not adaptive (i.e., the scoring function does not change as we observe new documents). Therefore, this approach does not benefit from the information that is captured as the extraction process progresses.

In this paper, we advocate an adaptive document ranking approach that addresses the above limitations of the state-of-the-art techniques. Specifically, we propose a principled, efficient learning-to-rank approach that prioritizes documents for an information extraction task by combining: (i) online learning [30], to train and adapt the ranking models incrementally, hence avoiding computationally expensive retrains of the models from scratch; and (ii) in-training feature selection [17], to identify a compact, discriminative set of words and phrases from the documents to train ranking models effectively and efficiently. Importantly, our approach revises the document ranking decisions periodically, as the ongoing extraction process reveals (fine-grained) characteristics of the useful documents for the extraction task at hand. Our approach thus manages to capture, progressively and in an adaptive manner, the heterogeneity of language and content typically exhibited by the useful documents, which in turn leads to information extraction executions that are substantially more efficient—and effective—than those with state-of-the-art approaches, as we will see. In summary, we present an end-to-end document ranking approach for effective and efficient information extraction in an adaptive, online, and principled manner. Our main contributions are:

- Two low-overhead ranking algorithms for information extraction based on learning-to-rank strategies. These algorithms perform online learning and in-training feature selection (Section 3.1).
- Two techniques to detect when adapting the ranking model for information extraction is likely to have a significantly positive impact on the ranking quality (Section 3.2).
- An experimental evaluation of our approach using multiple extraction tasks implemented with a variety of extraction approaches (Sections 4 and 5). Our approach has low overhead and manages to achieve higher accuracy than the state-of-the-art approaches, and hence is a substantial step towards scalable information extraction.

2. BACKGROUND AND RELATED WORK

Information extraction systems extract structured information from natural language text. For instance, an extraction system properly trained to extract tuples of an *Occurs-in*(*NaturalDisaster*, *Location*) relation might extract the tuple <tsunami, Hawaii> from the sentence: "*A tsunami swept the coast of Hawaii.*" These systems often rely on computationally expensive processing steps and, consequently, processing all documents exhaustively becomes prohibitively time consuming for large document collections [2]. Ideally, we should instead focus the extraction effort on the *useful* documents, namely, the documents that produce tuples when processed with the information extraction system at hand.

As a crucial task, information extraction optimization approaches (e.g., Holistic-MAP [31]) choose a document selection strategy to identify documents that are likely to be useful. State-of-the-art approaches for such document selection (e.g., QXtract [2], PRDualRank [14], and FactCrawl [7]) are based on the observation that useful documents for a specific relation[2] tend to share distinctive words and phrases. Discovering these words and phrases is challenging because: (i) many extraction systems rely on off-the-shelf, black-box components (e.g., named entity recognizers), from which we cannot extract relevant words and phrases directly; and (ii) machine learning techniques for information extraction do not generally produce easily interpretable models, which complicates the identification of relevant words and phrases. QXtract learns these words and phrases through document classification: after retrieving a small document sample, QXtract automatically labels each document as useful or not by running the extraction system of interest over these documents. QXtract can thus learn that words like "richter" or "hypocenter" are characteristic of some of the useful documents for *Occurs-in*. Then, QXtract uses these learned words and phrases as keyword queries to retrieve (other) potentially useful documents (see Figure 1). More recent approaches (e.g., FactCrawl [7] and PRDualRank [14]) adopt similar retrieval-based document selection strategies.

QXtract issues queries to the standard keyword search interface of document collections in order to retrieve potentially useful documents for extraction. Such keyword search interface, unfortunately, is not tailored for information extraction: the documents that are returned for a keyword query are ranked according to how well they match the query and not on how useful they are for the underlying information extraction task [7]. For example, the query [tornado] for the *Occurs-in* relation returns only 145 useful documents among the top-300 matches from our validation split of the New York Times annotated corpus[3] (see Section 4) using Lucene[4], a state-of-the-art search engine library.

FactCrawl [7] moves a step beyond keyword search and reranks the retrieved documents to prioritize the extraction effort (see Figure 1). Specifically, FactCrawl scores documents proportionally to the number and quality of the queries that retrieve them. FactCrawl determines the quality of each learned query—and of the query generation method that

[2]Our approach is not applicable over open information extraction scenarios (e.g., [4]) where most documents often contribute tuples to the open-ended extraction task.
[3]http://catalog.ldc.upenn.edu/LDC2008T19
[4]http://lucene.apache.org/

was used to generate the query—in an initial step, once and for all, by retrieving a small number of documents with the query and running them through the extraction system in question. With this initial step, FactCrawl derives: (i) for each query q, the F-measure $F_\beta(q)$, where β is a parameter that weights precision over recall; and (ii) for each query generation method m, the average $F_\beta^{avg}(m)$ of the F_β value of all queries generated with method m. During the extraction process, after retrieving documents with a set Q_d of queries learned via a query generation method m, FactCrawl re-ranks the documents according to a scoring function $S(d) = \sum_{q \in Q_d} F_\beta(q) \cdot F_\beta^{avg}(m)$. FactCrawl's document re-ranking process improves the efficiency of the extraction, since the documents more likely to be useful are processed earlier. However, FactCrawl exhibits two key weaknesses: (i) for document retrieval and ranking, just as QXtract (see discussion above), FactCrawl relies on queries derived, once and for all, from a small initial document sample, and hence may miss words and phrases relevant to the information extraction task at hand; and (ii) for document ranking, FactCrawl relies on a coarse, query-based document scoring approach that is not adaptive, and hence does not benefit from the wealth of information that is captured as the extraction process progresses.

Adaptive models have been used for information extraction in a variety of ways. Early influential systems for large-scale information extraction, such as DIPRE [10] and Snowball [1], have relied on bootstrapping to adapt to newly discovered information. Starting with a small number of "seed" tuples for the extraction task of interest, these systems learn and iteratively improve extraction patterns and, simultaneously, build queries from the tuples that they discover using these patterns. However, these systems are not suitable for our problem for two main reasons. First, techniques based on bootstrapping often exhibit far-from-perfect recall, since it is difficult to reach all tuples in a collection by using previously extracted tuples as queries [2, 19]. Second, extraction systems are many times "black box" systems, which impedes the alteration of their extraction decisions. Other approaches (e.g., [12]) have relied on label propagation: starting with labeled and unlabeled examples, these approaches propagate the given labels to the unlabeled examples based on some example similarity computation. Such label propagation approaches are not beneficial for our extraction scenario, where the extraction system has already been trained and we can obtain new labels (i.e., useful or not) for previously unseen documents automatically by running the extraction system over them.

3. ONLINE ADAPTIVE RANKING

We now propose an end-to-end document ranking approach for scalable information extraction (see Figure 2) that addresses the limitations of the state of the art. Our approach prioritizes documents for an information extraction task—with a corresponding already-trained information extraction system—based on principled, efficient learning-to-rank approaches that exploit the full contents of the documents (Section 3.1). Additionally, our approach revises the ranking decisions periodically as the extraction process progresses and reveals (fine-grained) characteristics of the useful documents for the extraction task at hand (Section 3.2). Our approach thus manages to capture, progressively and in an adaptive manner, the heterogeneity of language and

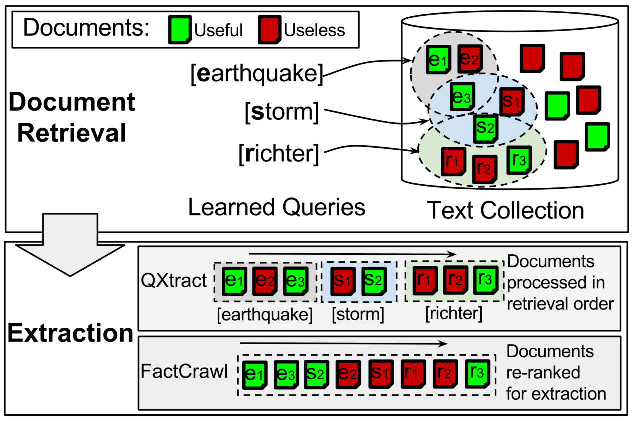


Figure 1: QXtract and FactCrawl.

content typically exhibited by the useful documents, which leads to extraction processes substantially more efficient—and effective—than those with state-of-the-art approaches, as we will show experimentally in Sections 4 and 5.

3.1 Ranking Generation

To prioritize the information extraction effort, by focusing on the potentially useful documents for the extraction system at hand, we follow a learning-to-rank approach (see Ranking Generation step in Figure 2). Similarly to state-of-the-art query-generation and ranking efforts (see Section 2), we obtain a small document sample and automatically "label" it with the information extraction system, without human intervention. We use the documents in this sample, with their words as well as the attribute values of tuples extracted from them as features, to train an initial document ranking model. After the initial document ranking is produced, we start processing documents, in order, with the information extraction system (see Tuple Extraction step in Figure 2).[5] Unfortunately, the initial ranking model is generally far from perfect, because it is learned from a necessarily small document sample. So our approach periodically updates and refines the ranking model (see Update Detection step in Figure 2), as new documents are processed and the characteristics of the useful documents are revealed, as we will discuss in detail in Section 3.2.

Unfortunately, state-of-the-art approaches for learning to rank [23] are problematic for our document ranking setting for two main reasons. First, such approaches tend to be computationally expensive [29], so updating and revising the ranking model continuously over time, as new documents are processed, would result in an unacceptably high overhead in the extraction process. Second, such approaches tend to require a relatively small feature space [3]. In contrast, in our ranking setting the feature space, including the document words and attributes of extracted tuples, is vast; furthermore, the feature space continues to grow as new documents are processed. Therefore, we need to develop unconventional learning-to-rank techniques for our ranking prob-

[5]The pool of documents to process is either the full document collection, for collections of moderate size over which we have full access, or, alternatively, the documents retrieved with queries learned from the document sample. In Sections 4 and 5, we discuss this issue further and experimentally study these two scenarios.

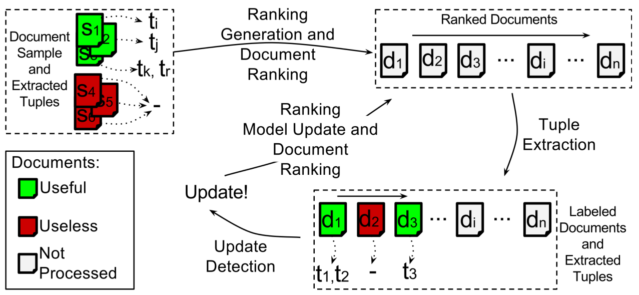

Document Sample and Extracted Tuples
S1
S2
S3
S4
S5
S6
ti
tj
tk, tr
-
Ranking Generation and Document Ranking
Ranked Documents
d1
d2
d3
...
di
...
dn
Tuple Extraction
Ranking Model Update and Document Ranking
Update!
Update Detection
Labeled Documents and Extracted Tuples
t1,t2
-
t3
Documents:
Useful
Useless
Not Processed

Figure 2: Our adaptive learning-to-rank approach for information extraction.

lem, to address the above two limitations of state-of-the-art approaches in an effective and efficient manner and without compromising the quality of the ranking models that we produce.

To address the efficiency limitation of learning-to-rank approaches, and to update the document ranking model efficiently, we rely on *online learning* [8]. Using online learning, we can train the ranking model incrementally, one document at a time. Therefore, we can continuously adapt the ranking model as we process new documents, without having to retrain it from scratch. To adapt online learning to our problem, the main challenge is to define an update rule for the model—to be triggered when we observe new documents along the extraction process—that is simple enough to be efficient but, at the same time, sophisticated enough to produce high-quality models. From among the most robust online learning approaches [8], the updates based on Pegasos gradient steps [30] are particularly well suited for our approach because of their efficiency and accuracy. Specifically, Pegasos gradient steps provide update rules that guarantee that learning techniques based on Support Vector Machines (SVM), the basis for some of the best-performing learning-to-rank approaches, learn high-quality models efficiently.

To address the feature-set limitation of learning-to-rank approaches, and to handle large (and expanding) feature sets, we rely on *in-training feature selection* [17]. In a nutshell, with in-training feature selection the learning-to-rank algorithm can efficiently identify the most discriminative features, out of a large and possibly expanding feature set, during the training of the document ranking model and without an explicit feature selection step. To do so, we rely on a sparse representation of the vectors that represent the feature weights, to discard all features with zero value. Therefore, our objective is to penalize models that rely on a large number of features with non-zero weight. Interestingly, we can rely on *regularization* [6] to control the feature weight distribution in our learned models: regularization penalizes models that have undesirable properties such as having many features with non-zero weights, so we can use it for in-training feature selection and also to avoid overfitting. In our approach, we rely on a linear combination of two regularization methods, usually called elastic-net regularization [35], which integrates: (i) the ℓ_1-norm regularization [32], which tends to learn models where only a small subset of the features have non-zero weights; and (ii) the ℓ_2-norm regularization, which produces high-quality models by avoiding overfitting. This combination is necessary because the ℓ_1-norm regularization does not perform well when the number of documents is smaller than the feature space [35], which is the case during early phases of the extraction process.

We now propose two learning-to-rank strategies, BAgg-IE and RSVM-IE, that overcome the limitations of state-of-the-art learning-to-rank approaches by integrating online learning and in-training feature selection, as discussed above.

BAgg-IE: Our first strategy incorporates online learning and in-training feature selection into a binary classification scheme where documents are ranked according to their assigned label and prediction confidence. Since binary classifiers optimize the accuracy of label assignment instead of the instance order, they are not optimized for ranking tasks [18]. For this reason, BAgg-IE adopts a more robust approach that exploits multiple binary classifiers based on bootstrapping aggregation, or bagging [9]. With this approach, the label assignments and confidence predictions derive from the aggregation of the answers of a committee of classifiers, rather than from an individual classifier. The intuition behind BAgg-IE is that each classifier is able to evaluate distinct aspects of the documents, thus collectively mitigating the limitations of each individual classifier. We adapt SVM-based binary classifiers [20] to support online learning and in-training feature selection. For online learning, our algorithm is based on Pegasos, in which each text document is a training instance and, hence, we update the model one document at a time. For in-training feature selection, each classifier in BAgg-IE combines the SVM binary classification problem with the regularization components of the elastic-net regularization framework that we discussed earlier, thus yielding the following learning problem to solve:

$$\arg\min_{\mathbf{w},b} \lambda_{All}\left(\frac{\lambda_{L2}}{2}\|\mathbf{w}\|_2 + (1-\lambda_{L2})\|\mathbf{w}\|_1\right) + \sum_{(\mathbf{d},y)\in\mathcal{S}} \ell(y\langle\mathbf{w},\mathbf{d}\rangle + b)$$

where b is the bias factor, ℓ is the hinge loss function, $\ell(t) = max(0, 1-t)$, and $\|\mathbf{w}\|_1$ and $\|\mathbf{w}\|_2$ are the ℓ_1 and ℓ_2-norms of the weight vector (i.e., the regularization components), respectively. Moreover, λ_{All} is the parameter that weights the regularization component over the loss function, and λ_{L2}, $0 \leq \lambda_{L2} \leq 1$, is the parameter that weights the ℓ_2-norm regularization over the ℓ_1-norm regularization.

The committee in BAgg-IE consists of three classifiers[6], trained over disjoint splits of the documents, which leads to different feature spaces for each, and with balanced labels (i.e., same number of useful and useless documents). Finally, to obtain the score of a text document we sum over the normalized scores of each classifier $s(\mathbf{d}) = \frac{1}{1+e^{-(\mathbf{w}^\top\mathbf{d}+b)}}$, which accounts for the differences in the feature weights of each classifier. In this equation, $\mathbf{w}$ and b are the weight vector and bias factor, respectively, of the classifier.

In summary, BAgg-IE addresses the ranking problem as an optimized classification problem. In contrast, our second technique, RSVM-IE, which we describe next, adopts a principled learning-to-rank approach natively.

RSVM-IE: Our second learning-to-rank strategy is based on RankSVM [21], a popular and effective pairwise learning-to-rank approach. Just as we did for BAgg-IE, we need to modify RankSVM's original optimization problem so that it incorporates in-training feature selection and, in turn, suits our ranking problem. In a nutshell, RankSVM scores the

[6]Additional classifiers would slightly improve performance at the expense of substantial overhead.

documents via a linear combination of the document features: the score of a document d is $s(d) = \sum_i w_i \cdot d_i$, where w_i is the weight of feature i and d_i is the value of feature i in document d. The objective of RankSVM is then to find the set of weights $\mathbf{w} = \{w_1, ..., w_n\}$ that is optimized to determine, in a pair of documents, if a document is more relevant than the other document. To achieve this, RankSVM learns the feature weights by comparing the features of useful and useless documents in pairs: each pair includes a useful and a useless document, and the label indicates whether the useful document is the first document in the pair.

By integrating the in-training feature selection discussed above into the original RankSVM formulation, we obtain the following optimization problem to solve for RSVM-IE:

$$\arg\min_{\mathbf{w}} \lambda_{All}(\frac{\lambda_{L2}}{2}\|\mathbf{w}\|_2+(1-\lambda_{L2})\|\mathbf{w}\|_1)+\sum_{(i,j)\epsilon\mathcal{P}} \ell(\mathbf{w}^\top(\mathbf{d}_i-\mathbf{d}_j))$$

where all variables are defined as for BAgg-IE, and $\mathbf{d}_i$ and $\mathbf{d}_j$ represent a useful and a useless document, respectively. For online learning, and in contrast to BAgg-IE, which uses the individual documents in the Pegasos scheme, the training examples are the pairs of useful and useless documents that the extraction process observes, which is known as *Stochastic Pairwise Descent* [29].

Unlike BAgg-IE, RSVM-IE is designed from the ground up to address a ranking task, so we expect it to outperform BAgg-IE. Moreover, we expect the overhead of RSVM-IE to be substantially lower than that of BAgg-IE, since BAgg-IE maintains multiple learned models (i.e., the classifiers in the committee). This overhead becomes noticeable when the models are frequently updated. Next, we explain our approach to decide when an update of the ranking models is desirable during the extraction process, thus reducing the overall document re-ranking overhead.

3.2 Update Detection

As we mentioned in Section 3.1, our adaptive extraction approach revises the ranking decisions periodically, to account for the new observations gathered along the extraction process. To determine when to update the ranking model (and, correspondingly, the document ranking), we introduce the *Update Detection* step (see Figure 2). To make this decision, we analyze whether the features of recently processed documents differ substantially from those in the ranking model. If this is the case, then we trigger a new ranking generation step (Section 3.1), which uses the recently processed documents as additional training examples. The new training examples often reveal novel features, or lead to adjusting the weight of known features, which in turn helps to more effectively prioritize the yet-unprocessed documents.

One possible approach for update detection is through feature shifting detection techniques [16]. Feature shifting predicts whether the distribution of features in a (test) dataset differs from the distribution of the features in the training data. Unfortunately, most feature shifting techniques are problematic: First, they rely on computationally expensive algorithms (e.g., kernel-based one-class SVM classifier [16]), thus incurring substantial overhead when applied repeatedly. Second, these techniques only detect changes in existing features, so they do not handle well the evolving feature space in our problem. Thus, the features that do not appear in the ranking would not be considered in the comparison, unless we re-train the kernel-based classifier from scratch, which would be prohibitively expensive.

As efficient alternatives, we introduce two update detection approaches, namely, Top-K and Mod-C. Top-K evaluates a reduced set of highly relevant features, determined independently from the ranking model, whereas Mod-C directly manipulates the low-level characteristics of the ranking model to detect changes in the feature space.

Top-K: Our first approach exploits the fact that the predicted usefulness of the documents in the current ranking varies the most when the highly influential features in the ranking model change. For instance, if the word "lava" becomes more frequent along the processed useful documents in our *Occurs-in* example, this feature will become (temporarily) more relevant than others. In that case, the predicted usefulness of documents that include such word should increase accordingly to be prioritized over other documents. Based on this observation, Top-K compares the K most influential features in the current ranking against the K most influential features according to the recently processed documents, and triggers an update when the difference between these two sets exceeds a given threshold τ, determined experimentally, as we explain in Section 4. Overall, Top-K consists of two key steps: (i) *feature selection*, which selects the K most influential features; and (ii) *feature comparison*, which measures the distance between two sets of features. To perform feature selection, we choose the K features with highest weight in an SVM-based linear classifier trained—and subsequently updated—on the same features (i.e., words and tuple attributes) as the ranking algorithm. To perform feature comparison, we compute a generalized version of the Spearman's Footrule[7] [22], which considers the relative position of the features and their weights. According to this measure, the difference between feature weights will be higher when heavily weighted features change positions.

As discussed, Top-K maintains its own set of relevant features according to an SVM-based binary classifier. The advantage of this approach is that it makes Top-K independent of the ranking technique. However, the relevant features in this classifier may differ from those in the ranking model [18]. In our *Occurs-in* example, for instance, a trained RankSVM model weighted the word "northern" as a top-20 feature, whereas a linear SVM model trained on the same documents weighted "northern" almost neutrally. Such discrepancies in the feature relevance may cause updates that have little impact on the document ranking or, alternatively, may lead to missing necessary updates because important features are not being evaluated. We now introduce Mod-C, which works directly with the ranking models, to capture feature relevance directly.

Mod-C: The techniques in Section 3.1 learn ranking models that consist of a vector of numeric weights, where each weight represents the captured relevance of one feature. We can then use a vector similarity metric, such as cosine similarity [24], to measure the difference between the relevance of features in two similar ranking models. Our second technique, Mod-C, exploits this observation and compares the current ranking model to an "updated" ranking model that also includes some of the recently processed documents. This

[7]The generalized version of the Spearman's Footrule that we use is given by $\sum_i w_i \cdot \left|\sum_{j:j\leq i} w_j - \sum_{j:\sigma(j)\leq\sigma(i)} w_j\right|$, where $\sigma(i)$ is the rank of feature i and w_i is its weight.

updated ranking model includes only a fraction ρ of the recently processed documents, since including all of these documents would incur substantial overhead. To compare the ranking models, Mod-C depends on a metric suitable for the ranking model (e.g., cosine similarity for RSVM-IE) and a threshold α, determined experimentally as we explain in Section 4, that needs to be exceeded to trigger an update. In our cosine similarity example, α would indicate the maximum allowed angle between ranking models, hence triggering an update when this angle is exceeded. Mod-C is thus able to handle the real relevance of features, crucial to precisely decide when an update in the ranking model will improve the current document ranking.

In summary, we propose two update detection techniques that decide efficiently when it is beneficial to revise the ranking decisions to adaptively improve the extraction process.

4. EXPERIMENTAL SETTINGS

We now describe the experimental settings for the evaluation of our adaptive ranking approach:

Datasets: We used the *NYT Annotated Corpus* [28], with 1.8 million New York Times articles from 1987 to 2007. We split this corpus into a training set (97,258 documents), a development set (671,457 documents), and a test set (1,086,944 documents). We evaluated different combinations of techniques and parameters on the development set. We ran the final experiments on the test set. Additionally, we used collections 1-5 from the TREC conference[8] to generate the queries for the query-based sample generation that we explain later in this section.

Document Access: As mentioned in Section 3.1, we consider two document-access scenarios: In the *full-access* scenario, we rank all documents in a (moderately sized) document collection. In contrast, in the (more realistic) *search interface access* scenario, we retrieve the documents to rank through keyword queries. We evaluate our ranking approach over both scenarios. For the search interface access scenario we learn the queries following QXtract (Section 2) to retrieve an initial pool of documents. Also, we provide a search interface over our collection using the Lucene indexer, to retrieve additional documents as the extraction process progresses: after each ranking update, we use the top-100 features of the updated ranking model as individual text queries to retrieve additional (potentially) useful documents.

Relations: Table 1 shows the broad range of relations from different domains that we extract for our experiments, with the number of useful documents for each relation in the test set. Our relations include *sparse* relations, for which a relatively small fraction of documents (i.e., less than 2% of the documents) are useful, as well as *dense* relations.

Information Extraction Techniques: We selected the extraction approach for each relation to include a variety of extraction approaches (e.g., both machine learning and rule-based approaches, as well as techniques with varying speed). Specifically, we considered different entity and relation extractors for each relation, and selected the best performing combination. However, for diversity, whenever we had ties in performance, we selected the (arguably) less common contender (e.g., a pattern-based approach to extract organizations and Maximum Entropy Markov Model [25], or MEMM, for natural disasters):

[8]http://trec.nist.gov/data.html

Relation	Useful Documents
Person–Organization Affiliation (PO)	185,237 (16.95%)
Disease–Outbreak (DO)	847 (0.08%)
Person–Career (PC)	458,294 (42.16%)
Natural Disaster–Location (ND)	18,370 (1.69%)
Man Made Disaster–Location (MD)	15,837 (1.46%)
Person–Charge (PH)	19,237 (1.77%)
Election–Winner (EW)	5,384 (0.50%)

Table 1: Relations for our experiments.

- For the Person–Organization Affiliation relation we used Hidden Markov Models [13] and automatically generated patterns [34] as named entity recognizers for Person and Organization, respectively. We used SVM [15] to extract the relation.
- For the Disease–Outbreak relation we used dictionaries and manually crafted regular expressions as named entity recognizers for Disease and Temporal Expression, respectively. We used the distance between entities to predict if they are related.
- For the remaining relations, we used Stanford NER[9] to find Person and Location entities, a MEMM [25] to find Natural Disasters, and Conditional Random Fields [26] to find the remaining entities. Then, we used the Subsequence Kernel [11] to identify relations between these entities.

Development Toolkits: We used the following off-the-shelf libraries: (i) Lingpipe[10], for rule-based named entity extraction; (ii) OpenNLP[11], for word and sentence segmentation; (iii) E-txt2db[12] and Stanford NER, to train and execute named entity extractors based on machine learning; and (iv) REEL[13] [5], to train relation extraction models.

Sampling Strategies: We compared two techniques to collect the initial document sample for our ranking techniques (Section 3.1):

- *Simple Random Sampling (SRS):* SRS picks 2,000 documents at random from the collection (only for the *full-access* scenario).
- *Cyclic Query Sampling (CQS):* CQS iterates repeatedly over a list of queries and collects the unseen documents from the next K documents that each query retrieves until it collects 2,000 documents. We learned 5 lists of queries using sets of 10,000 random documents (5,000 useful and 5,000 useless) from the TREC collection by applying the SVM-based method in QXtract [2].

Ranking Generation Techniques: We evaluated our ranking generation techniques from Section 3.1. To obtain the best parameters for these techniques, we performed several experiments over our development set, varying λ_{All} and λ_{L2}. The parameter values that we determined experimentally are as follows: for *BAgg-IE*, $\lambda_{All} = 0.5$ and $\lambda_{L2} = 0.99$; while for *RSVM-IE*, $\lambda_{All} = 0.1$ and $\lambda_{L2} = 0.99$. Setting $\lambda_{L2} = 0.99$ results in an ℓ_1-norm weight of $1 - \lambda_{L2} = 0.01$. This weight in turn results in models with 10 times fewer features—which are hence 10 times faster—than models that only use the ℓ_2-norm. Higher ℓ_1-norm weights would lead to

[9]http://nlp.stanford.edu/software/CRF-NER.shtml
[10]http://alias-i.com/lingpipe/
[11]http://opennlp.apache.org/
[12]http://web.ist.utl.pt/ist155840/etxt2db/
[13]http://reel.cs.columbia.edu/

lower-quality ranking models, as discussed in Section 3.

We also evaluated the following (strong) baselines:

- *FactCrawl (FC):* FC corresponds to our implementation of FactCrawl [7], as described in Section 2.
- *Adaptive FactCrawl (A-FC):* We produced a new version of FC that re-ranks the documents. Specifically, to make FC more competitive with our adaptive ranking strategies, A-FC recomputes the quality of the queries, and re-ranks the documents with these new values after each document is processed. In addition, A-FC learns new queries and retrieves more documents before every re-ranking step.

(We evaluated other approaches, such as QXtract [2] and PRDualRank [14], but do not discuss them further because FactCrawl dominated the alternatives that we considered.)

Update Detection Techniques: We evaluated our update detection techniques from Section 3.2:

- *Top-K:* We set $K = 200$, which experimentally led to high coverage of the relevant features and small overhead in feature comparison. We set $\tau = \varepsilon \cdot K$, where ε indicates how much each feature can change without impacting the ranking. We experimented with several values of τ and finally picked $\tau = 0.5$ ($\varepsilon = 0.0025$).
- *Mod-C:* We evaluated several combinations of ρ and α: the best value for ρ is 0.1, while the best angle values for α are 5° and 30° for RSVM-IE and BAgg-IE, respectively.

We also compared against the following baselines:

- *Wind-F:* We implemented a naïve approach for update detection that updates the ranking model after processing a fixed number of documents. We experimented with several values and observed no substantial differences. We report our results for updating 50 times along the extraction process, which leads to updates after 13,429 and 21,739 documents for the validation and test sets, respectively.
- *Feat-S:* We implemented an efficient version of feature shifting [16] using an online one-class SVM based on Pegasos [30]. We used a Gaussian kernel with $\gamma = 0.01$ and $k = 6$, as suggested in [16]. Finally, we triggered an update when the geometrical difference $F = 1 - S$ exceeded a threshold $\tau = 0.55$. Since the features of the documents after each update tend to fluctuate, we only run Feat-S after processing 700 new documents or more.

Executions and Infrastructure: We ran all experiments over a cluster with 60 machines with a uniform configuration: Intel Core i5-3570 CPU @ 3.40 GHz processors, with 8 GB of RAM, and OS Debian GNU/Linux 7 (wheezy). We used multiple independent processes to test our approach with different configurations. We executed each experiment five times with different samples (i.e., five different random samples and five different sets of initial sample queries), to account for the effect of randomness in the results, and report the average of these executions.

Evaluation Metrics: We use the following metrics:

- *Average recall* is the recall of the extraction process (i.e., the fraction of useful documents in the collection that have been processed) at different points during the extraction (e.g., after processing $x\%$ of the documents) and averaged over all executions of the same configuration.
- *Average precision* is the mean of the precision values at every position of the ranking [33], averaged over all the executions of the same configuration.
- *Area Under the ROC (AUC)* is the area under the curve of the true positive rate as a function of the false positive rate, averaged over all the executions of the same configuration.

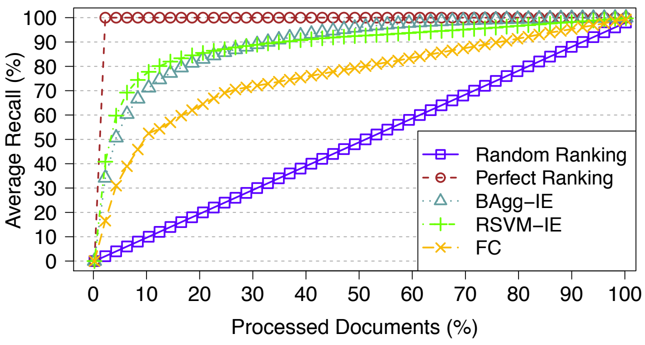


Figure 3: Average recall for Person–Charge for different base ranking generation techniques.

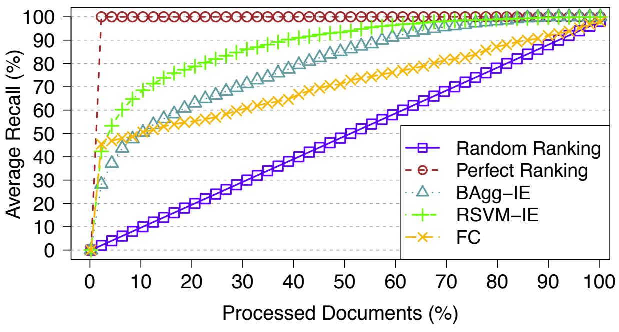


Figure 4: Average recall for Disease–Outbreak for different base ranking generation techniques.

- *CPU time* measures the CPU time consumed for extracting and ranking the documents.

5. EXPERIMENTAL RESULTS

We now present the results of the experimental evaluation of our adaptive ranking approach. We tuned the configuration of all components of our approach (i.e., the sampling strategy, the learning-to-rank approach, and the update detection approach) by exhaustively considering all possible combinations over the development set and selecting the best such combination. In the discussion below, for clarity, we consider the configuration choices for each component separately. Later, for the final evaluation of our approach over the test set and against the state-of-the-art ranking strategies, we use the best configuration according to the development set experiments.

Impact of Learning-To-Rank Approach: To understand the impact of using our learning-to-rank approach, we first evaluate our techniques of Section 3.1, *without the adaptation step*, against FC over the development set. Figure 3 shows the average recall for the Person–Charge relation for the full-access scenario. (For reference, we also show the performance of a random ordering of the documents, as well as of a perfect ordering where all useful documents are ahead of the useless ones.) Both RSVM-IE and BAgg-IE consistently outperform FC. Interestingly, RSVM-

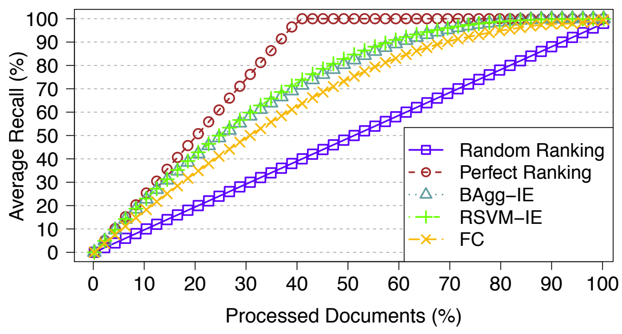


Figure 5: Average recall for Person–Career for different base ranking generation techniques.

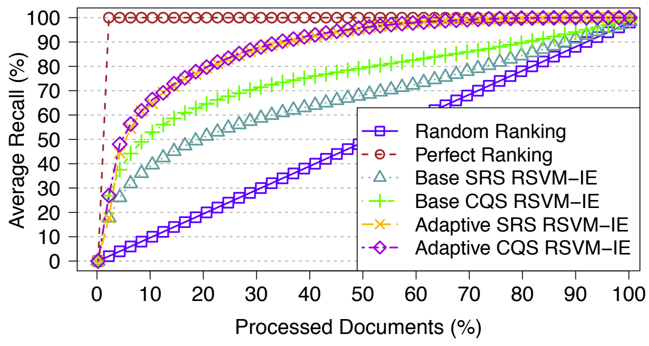


Figure 6: Average recall for Man Made Disaster–Location with different sampling techniques for the base and adaptive versions of RSVM-IE.

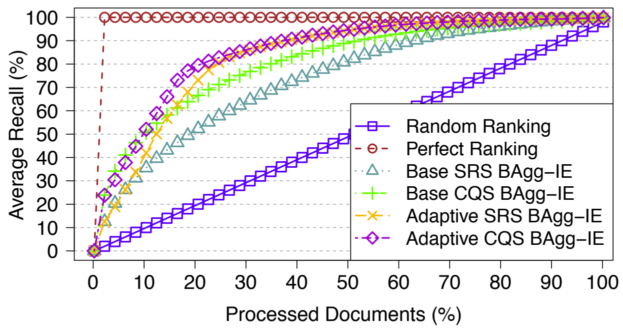


Figure 7: Average recall for Man Made Disaster–Location with different sampling techniques for the base and adaptive versions of BAgg-IE.

IE performs better in early phases of the extraction, while BAgg-IE performs better in the later phases, which agrees with our intuition from Section 3.1: RSVM-IE is at its core a ranking optimization technique, while BAgg-IE is based on classifiers. BAgg-IE separates useful from useless documents, thus obtaining high-accuracy in the middle of the extraction process, which in turn leads to high recall later on. We observed similar results for most of the relations (e.g., Figures 4 and 5 show the results for Disease–Outbreak and Person–Career respectively). However, RSVM-IE performs better than BAgg-IE for sparse relations, so RSVM-IE is preferable for such relations even in later phases of the extraction process (see Figure 4). Overall, even without an adaptation step, our techniques outperform the state-of-the-art ranking technique FC.

Impact of Sampling Strategies: To understand the impact of different sampling techniques to learn the initial ranking model, we compared RSVM-IE and BAgg-IE using the SRS and CQS sampling techniques (Section 4). Figure 6 shows the average recall for the Man Made Disaster–Location relation in the full-access scenario for RSVM-IE, both without the adaptation step (denoted with keyword "Base" in the plot) as well as with adaptation (denoted with keyword "Adaptive"). (The results for BAgg-IE were analogous; see Figure 7.) Using CQS, a sophisticated sampling technique, has a generally positive impact relative to using the (simpler) SRS strategy. The only exceptions were the dense relations, namely, Person–Organization and Person–Career, for which a simple random sample typically includes a wide variety of useful documents, thus leading to high-quality models.

Impact of Adaptation: We claimed throughout this paper that refining the document ranking along the extraction process significantly improves its efficiency. To support this claim, Figure 6 shows the average recall of RSVM-IE for the Man Made Disaster–Location relation for the full-access scenario. (The results for BAgg-IE are analogous, although the difference between the sampling techniques is higher than for RSVM-IE; see Figure 7.) These results show that by adapting the ranking model learned by RSVM-IE and, correspondingly, the document ranking, we significantly improve the efficiency of the extraction process. For example, Figure 6 shows that the adaptive versions of RSVM-IE can reach 70% of the useful documents after processing only 10% of the collection, whereas the base (non-adaptive) versions only reached 40% and 50% of the useful documents, for SRS and CQS, respectively. This same behavior was replicated by almost all relations. Additionally, as shown in Figure 6, the sampling technique does not have a significant impact anymore when we incorporate the adaptation step. Nevertheless, we observed that the results of average precision and AUC (see Table 2) are generally better for CQS than for SRS, since CQS leads to processing more useful documents at early stages of the extraction process.

Finally, we evaluated the number of new features incorporated into the ranking model during the adaptation step. In early stages of the extraction process, an average of 200 (or about 25% of the total number of features in the previous models) are incorporated; a similar number of features is removed in each adaptation step. However, in later stages, this behavior changes as the models become more stable. Specifically, the number of incorporated and removed features drops to 10 after each adaptation step. These results show that while the initial adaptation steps significantly impact the ranking model, the later ones are insignificant. Therefore, it is important to properly schedule the adaptation step to avoid insignificant updates to the ranking model.

Impact of Update Detection: To evaluate the update detection techniques that we introduced in Section 3.2, we fix the document sampling to SRS, and evaluate the tech-

	Base SRS		Base CQS		Adaptive SRS		Adaptive CQS	
Rel.	A. Precision	AUC	A. Precision	AUC	A. Precision	AUC	A. Precision	AUC
PO	33.6±0.9%	76.7±1.0%	37.9±1.0%	77.7±0.9%	**44.2±0.3%**	**82.7±0.1%**	43.6±0.3%	**82.7±0.1%**
DO	2.3±1.1%	88.2±2.2%	3.1±0.6%	87.9±0.9%	3.0±1.0%	97.0±0.1%	**3.8±0.6%**	**97.1±0.1%**
PC	80.2±0.4%	86.9±0.2%	79.2±0.5%	86.5±0.4%	**84.2±0.2%**	**89.9±0.1%**	84.1±0.2%	**89.9±0.1%**
ND	6.1±1.1%	64.0±3.5%	13.1±0.9%	64.3±3.2%	10.2±0.9%	**85.5±0.2%**	**16.4±0.8%**	85.4±0.2%
MD	7.3±1.8%	67.4±3.2%	13.6±1.2%	76.6±3.6%	12.9±1.4%	88.6±0.2%	**17.2±0.8%**	**89.2±0.1%**
PH	28.6±0.7%	89.7±1.4%	28.1±1.1%	87.3±1.6%	33.0±0.6%	**95.5±0.0%**	**33.4±0.6%**	95.4±0.0%
EW	6.6±4.0%	79.5±8.6%	10.2±0.8%	84.6±1.4%	9.4±3.2%	94.9±0.5%	**12.6±0.6%**	**95.3±0.1%**

Table 2: Comparison of the impact of different document sampling techniques on the ranking quality for all the relations with the base and adaptive versions of RSVM-IE for the full-access scenario.

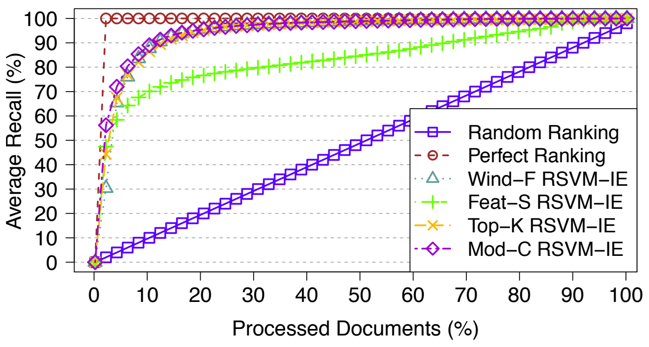


Figure 8: Average recall for Election–Winner for different update methods with RSVM-IE.

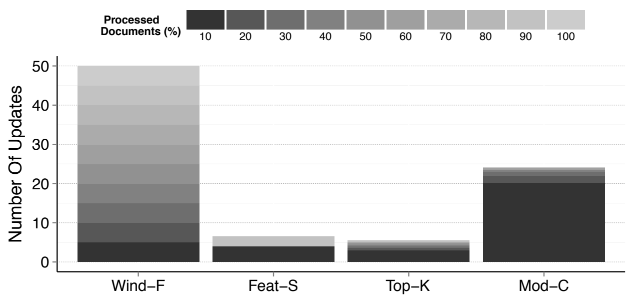


Figure 9: Distribution of updates for different techniques over the Election–Winner relation with RSVM-IE. (Darker shades represent earlier stages of the extraction process.)

Update Technique	CPU Time per Document
Wind-F	0.01±0.00 ms
Feat-S	5.72±0.29 ms
Top-K	1.89±0.71 ms
Mod-C	0.32±0.10 ms

Table 3: Average CPU time to perform update detection after processing each document.

niques according to their impact on the extraction process, distribution of updates, and overhead. Figure 8 shows the results of RSVM-IE for the Election–Winner relation for the full-access scenario. (The behavior for the other relations is analogous.) The Feat-S technique performed poorly in comparison to others, because Feat-S stops performing updates when the features observed in the data stabilize with respect to its kernel-based definition of shifting. For this reason, Feat-S misses late updates that prioritize other still poorly ranked useful documents. In addition, we observe that both Top-K and Mod-C produce consistently better results than Wind-F, especially at early stages of the extraction process, thus leading to high recall early in the extraction process. Overall, we show that both Top-K and Mod-C are robust alternatives for update detection in terms of ranking quality.

We also studied the distribution of updates across the extraction process, to understand the behavior of Top-K and Mod-C. Figure 9 shows the number of updates that each technique performs at different stages of the extraction process. Top-K and Mod-C tend to update much more frequently in early stages, where almost all documents carry new evidence of usefulness, than in later stages. For instance, most of the updates are performed while processing the first 10% of the collection. This behavior leads to ranking models that stabilize soon, since they are able to overcome the usual lack of training data in the initial document samples. Interestingly, despite the density of updates early in the process, the overall number of updates of Top-K and Mod-C remains smaller than that of Wind-F, since our techniques avoid unnecessary updates in late phases of the extraction process.

Additionally, we observed the percentage of features that are added to and eliminated from the models: the adaptation steps triggered by Top-K and Mod-C incorporate a consistent percentage of new features (i.e., about 10% per adaptation step) throughout the extraction process. This behavior significantly differs from that of Wind-F, which incorporates a large fraction of new features in early phases of the extraction process but only a small fraction of features later on: Top-K and Mod-C only perform an update when it will have a significantly positive impact on the model.

Finally, to evaluate the impact on efficiency of the update detection techniques, we calculated the overhead per document in terms of average CPU time, which we summarize in Table 3. As expected, Wind-F incurs negligible overhead (roughly 0.01 ms per document), since it only keeps a counter of the processed documents, whereas Feat-S incurs the highest overhead (5.72 ms per document). Our two techniques, Top-K (1.89 ms per document) and Mod-C (0.32 ms per document), exhibit a substantial difference in terms of efficiency, since the overhead of Top-K is dominated by the use of the binary classifier, as we discussed in Section 3.2. In conclusion, and considering also the quality results, Mod-C consistently outperforms the other techniques.

Scalability of our Approach: To understand how our

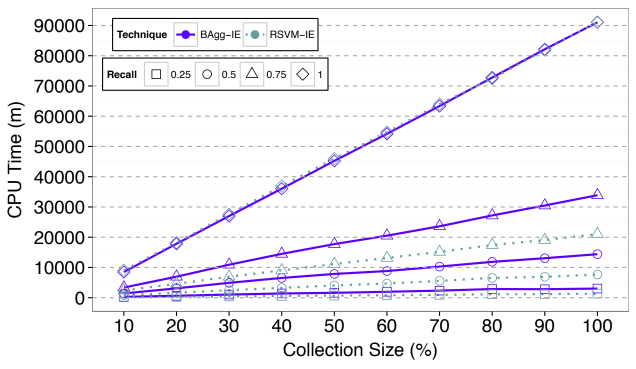


Figure 10: Average CPU time of our techniques as a function of the collection size for different target recall values, for the Natural Disaster–Location relation.

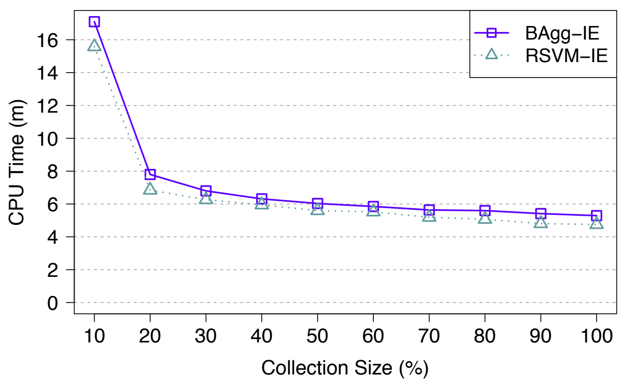
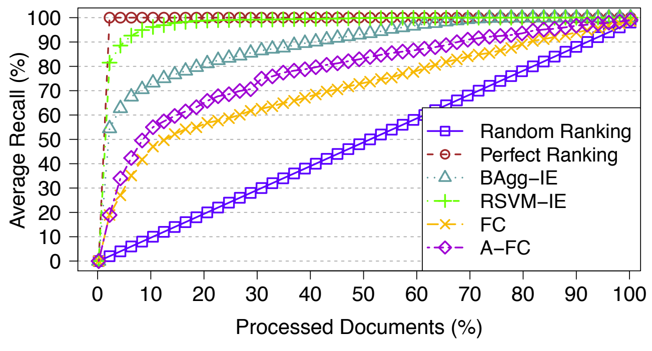


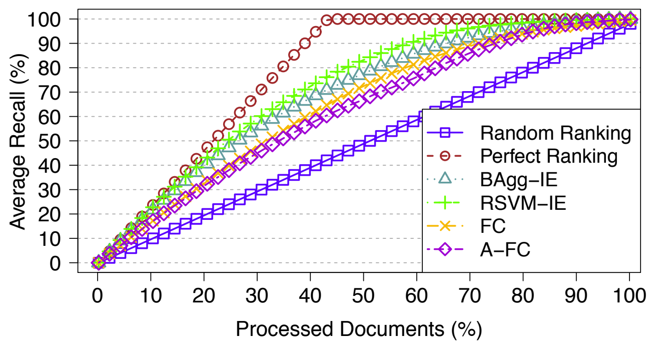
Figure 11: Average CPU time to find a target number of documents (i.e., the number of useful documents in the subset with 10% of the collection) for the Person–Organization Affiliation relation, as a function of the collection size.

strategies scale with the document collection size, we produced 10 subsets of the test collection with different sizes (from 10% to 100% of the total collection) and we measured (i) the time overhead for producing the ranking and (ii) the extraction time needed to reach a (fixed) target number of useful documents in each subset. Figure 10 shows how the size of the collection affects the CPU time needed to perform the ranking and extraction tasks with our techniques for the Natural Disaster–Location relation: the CPU time needed to perform an extraction task with our techniques grows approximately linearly with the collection size, which is desirable. Additionally, Figure 11 shows—for the Person–Organization Affiliation relation—that the time needed to find and process a target number of useful documents significantly drops as we increase the size of the collection. In this figure, the target number of useful documents corresponds to that in the subset of the collection that only contains 10% of the documents. As shown, the time becomes almost constant when the number of useful documents in the subset is large enough for the ranking to reach the target number at very early phases of the extraction process.

Comparison with State-of-the-Art Ranking Strategies: We now compare our best performing ranking approaches with the state-of-the-art approaches discussed in Section 4. We selected the best configuration for RSVM-IE and BAgg-IE according to the previous experiments, which involve CQS sampling and Mod-C update detection. Then, we ran this configuration over the test set to compare with FC and A-FC. We performed this experiment in the search interface access scenario as well, with similar conclusions. We compare the techniques on ranking quality and efficiency.



(a) Disease–Outbreak



(b) Person–Career

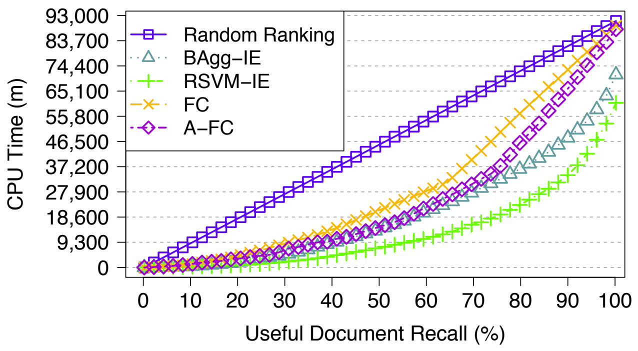
Figure 12: Average recall for different ranking approaches in the full-access scenario.

Table 4 shows the average precision and AUC of the four techniques that we compare, for all relations and over the full access scenario: RSVM-IE and BAgg-IE generally outperform the FactCrawl baselines by a large margin, and RSVM-IE consistently outperforms BAgg-IE. Interestingly, our adaptive version of FactCrawl, A-FC, does not exhibit the same significant improvement compared to FC that we observed between the adaptive and base versions of RSVM-IE and BAgg-IE above: A-FC is unable to properly model the usefulness of the documents when new features emerge, since it only relies on a small number of features.

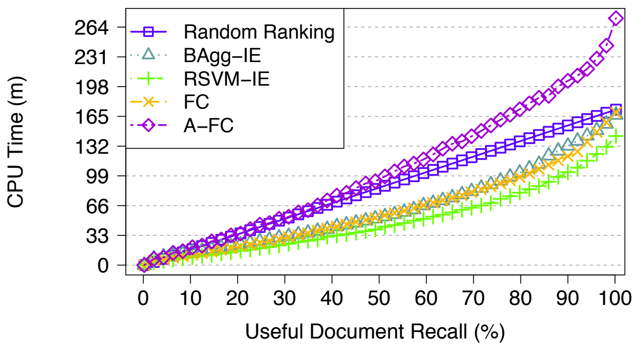
To understand the effects of the relation characteristics, we studied the performance of the techniques over both sparse (Figure 12a) and dense (Figure 12b) relations. The performance gap is more evident for sparse relations than it is for dense relations: The vocabulary around mentions of sparse relations tends to be reduced and specific, which makes it easier to model and prioritize the useful documents. Conversely, dense relations are scattered across diverse documents, thus co-occurring with a large variety of words, which makes it difficult to select a set of features that

	BAgg-IE		RSVM-IE		FC		A-FC	
Rel.	A. Precision	AUC	A. Precision	AUC	A. Precision	AUC	A. Precision	AUC
PO	40.5±0.9%	78.2±0.6%	**45.7±0.3%**	**82.4±0.1%**	29.0±0.9%	68.9±0.5%	30.5±0.6%	71.9±0.8%
DO	3.5±1.3%	89.7±0.3%	**8.3±0.2%**	**98.2±0.1%**	1.5±0.4%	71.5±11.4%	1.6±0.4%	78.8±5.4%
PC	79.2±0.4%	83.7±0.4%	**85.1±0.1%**	**88.6±0.1%**	66.3±1.1%	76.3±0.4%	63.2±1.0%	72.9±0.5%
ND	10.2±1.4%	78.4±0.5%	**18.9±0.6%**	**85.8±0.1%**	6.0±0.4%	67.8±1.5%	7.1±0.4%	72.9±0.2%
MD	10.8±2.1%	81.4±1.2%	**17.0±0.1%**	**88.0±0.0%**	3.8±0.4%	67.1±1.7%	4.1±0.4%	69.9±1.5%
PH	22.3±2.6%	90.5±2.1%	**33.8±0.3%**	**95.1±0.0%**	10.0±1.5%	74.6±2.8%	11.0±1.2%	78.9±1.5%
EW	9.6±0.6%	90.2±0.2%	**15.5±0.3%**	**95.4±0.1%**	2.4±0.2%	78.1±1.5%	2.6±0.2%	80.5±1.3%

Table 4: Comparison of the rankings generated by different techniques for the full-access scenario.



(a) Natural Disaster–Location



(b) Person–Organization Affiliation

Figure 13: CPU time to obtain a target recall value.

precisely identifies useful documents. Regardless, RSVM-IE and BAgg-IE still outperform the other techniques, since they are able to handle feature spaces of variable sizes.

We evaluate efficiency by measuring the time—including both ranking and extraction time—that each technique requires to achieve different values of recall. We show the results for two relations that exhibit substantially different extraction times according to their respective information extraction system: (i) Natural Disaster–Location, which takes an average of 6 seconds per document (Figure 13a); and (ii) Person–Organization Affiliation, which takes an average of 0.01 seconds per document (Figure 13b). RSVM-IE outperforms the others, in agreement with our earlier findings. The results for Person–Organization Affiliation are, in contrast, slightly different. For this fast extraction task, the overhead of the ranking technique can be problematic since it may easily become larger than the extraction time per se. We can observe such behavior for A-FC, which is less efficient than a random ranking technique with no overhead: A-FC (and, correspondingly, FC) relies on features that are expensive to compute [7], which is problematic for the adaptive case. However, the other techniques behave similarly as for the more expensive relations, with RSVM-IE resulting in the most efficient extraction process. Interestingly, for extraction tasks that incur lengthier extraction time, as is the case for Natural Disaster–Location, the quality of the ranking has a higher impact on efficiency than for other extraction tasks.

Overall, our experiments show that RSVM-IE outperforms all other techniques in all settings and extraction tasks. More specifically, RSVM-IE produces better rankings, while incurring very little overhead. Finally, when combined with Mod-C, RSVM-IE achieves much lower extraction times than the alternative strategies that we studied. Indeed, even with fast information extraction systems, adaptively ranking documents with RSVM-IE remained the best choice. Additionally, we evaluated the scalability of our techniques and confirmed that as the size of the collection grows, so does the positive impact of our approach, making it a substantial step towards scalable information extraction.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we presented an adaptive, lightweight document ranking approach for information extraction. Our approach enables effective and efficient information extraction over large document collections. Specifically, our approach relies on learning-to-rank techniques that learn in a principled way the fine-grained characteristics of the useful documents for an extraction task of interest. Our techniques incorporate (i) online learning algorithms, to enable a principled, efficient, and continuous incorporation of new relevant evidence as the extraction process progresses and reveals the real usefulness of documents; and (ii) in-training feature selection, to enable the learning of ranking models that rely on a small, discriminative set of features. Our experiments show that our approach exhibits higher recall and precision than state-of-the-art approaches, while keeping the overhead low. Overall, our document ranking approach is a substantial step towards scalable information extraction.

As future work, we plan to study how to estimate the recall of the alternative document ranking approaches for an information extraction task of interest. Through such estimates, we could in turn estimate the extraction cost, as a function of the number of processed documents, to achieve a target recall value with each ranking approach. We could then explore the recall-extraction cost tradeoff in a robust, quantitative manner, and substantially enhance recent optimization efforts for information extraction programs (e.g., [31]) by integrating our approach as an alternative document selection technique. As another direction for future work, we plan to continue studying our document ranking approaches along other dimensions, so that, for example, we can characterize ranking models according to the diversity of the tuples that they tend to produce. Finally, we aim at exploring parallelization approaches that, combined with the ranking-based approach described in this paper, can further speed up the execution of information extraction systems over large volumes of text data.

Acknowledgments: This material is based upon work supported by the National Science Foundation under Grant IIS-08-11038 and by *Fundação para a Ciência e a Tecnologia*, under Projects UID/CEC/50021/2013 and EXCL/EEI-ESS /0257/2012 (Datastorm), and Ph.D. Grant SFRH/BD /61393/2009. This work was also supported by the Intelligence Advanced Research Projects Activity (IARPA) via Department of Interior National Business Center (DoI/NBC) contract number D11PC20153. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of IARPA, DoI/NBC, or the U.S. Government.

7. REFERENCES

[1] E. Agichtein and L. Gravano. Snowball: Extracting relations from large plain-text collections. In *ACM-DL*, 2000.

[2] E. Agichtein and L. Gravano. Querying text databases for efficient information extraction. In *ICDE*, 2003.

[3] B. Bai, J. Weston, D. Grangier, R. Collobert, K. Sadamasa, Y. Qi, O. Chapelle, and K. Weinberger. Learning to rank with (a lot of) word features. *Information Retrieval*, 13(3):291–314, 2010.

[4] M. Banko, M. J. Cafarella, S. Soderland, M. Broadhead, and O. Etzioni. Open information extraction from the web. In *IJCAI*, 2007.

[5] P. Barrio, G. Simões, H. Galhardas, and L. Gravano. REEL: A relation extraction learning framework. In *JCDL*, 2014.

[6] C. M. Bishop. *Pattern recognition and machine learning*. Springer-Verlag, 2006.

[7] C. Boden, A. Löser, C. Nagel, and S. Pieper. FactCrawl: A fact retrieval framework for full-text indices. In *WebDB*, 2011.

[8] L. Bottou. Large-scale machine learning with stochastic gradient descent. In *COMPSTAT*, 2010.

[9] L. Breiman. Bagging predictors. *Machine Learning*, 24:123–140, 1996.

[10] S. Brin. Extracting patterns and relations from the world wide web. In *WebDB*, 1998.

[11] R. C. Bunescu and R. J. Mooney. Subsequence kernels for relation extraction. In *NIPS*, 2005.

[12] J. Chen, D. Ji, C. L. Tan, and Z. Niu. Relation extraction using label propagation based semi-supervised learning. In *COLING/ACL*, 2006.

[13] A. Ekbal and S. Bandyopadhyay. A Hidden Markov Model based named entity recognition system: Bengali and Hindi as case studies. *Lecture Notes in Computer Science*, 4815:545–552, 2007.

[14] Y. Fang and K. C.-C. Chang. Searching patterns for relation extraction over the web: Rediscovering the pattern-relation duality. In *WSDM*, 2011.

[15] C. Giuliano, A. Lavelli, and L. Romano. Exploiting shallow linguistic information for relation extraction from biomedical literature. In *EACL*, 2006.

[16] A. Glazer, M. Lindenbaum, and S. Markovitch. Feature shift detection. In *ICPR*, 2012.

[17] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *The Journal of Machine Learning Research*, 3:1157–1182, 2003.

[18] R. Herbrich, T. Graepel, and K. Obermayer. Large margin rank boundaries for ordinal regression. In *Advances in Large Margin Classifiers*. MIT Press, 2000.

[19] P. G. Ipeirotis, E. Agichtein, P. Jain, and L. Gravano. To search or to crawl?: Towards a query optimizer for text-centric tasks. In *SIGMOD*, 2006.

[20] T. Joachims. Text categorization with support vector machines: Learning with many relevant features. In *ECML*, 1998.

[21] T. Joachims. Optimizing search engines using clickthrough data. In *SIGKDD*, 2003.

[22] R. Kumar and S. Vassilvitskii. Generalized distances between rankings. In *WWW*, 2010.

[23] T.-Y. Liu. Learning to rank for information retrieval. *Foundations and Trends in Information Retrieval*, 3:225–331, 2009.

[24] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[25] A. McCallum, D. Freitag, and F. C. N. Pereira. Maximum entropy Markov models for information extraction and segmentation. In *ICML*, 2000.

[26] A. McCallum and W. Li. Early results for named entity recognition with conditional random fields, feature induction and web-enhanced lexicons. In *CONLL*, 2003.

[27] C. J. V. Rijsbergen. *Information Retrieval*. Butterworth-Heinemann, 2nd edition, 1979.

[28] E. Sandhaus. The New York Times Annotated Corpus. In *Linguistic Data Consortium*, 2008.

[29] D. Sculley. Large scale learning to rank. In *NIPS Workshop on Advances in Ranking*, 2009.

[30] S. Shalev-Shwartz, Y. Singer, and N. Srebro. Pegasos: Primal Estimated sub-GrAdient SOlver for SVM. In *ICML*, 2007.

[31] G. Simões, H. Galhardas, and L. Gravano. When speed has a price: Fast information extraction using approximate algorithms. In *PVLDB*, 2013.

[32] Y. Tsuruoka, J. Tsujii, and S. Ananiadou. Stochastic gradient descent training for L1-regularized log-linear models with cumulative penalty. In *ACL*, 2009.

[33] A. Turpin and F. Scholer. User performance versus precision measures for simple search tasks. In *SIGIR*, 2006.

[34] C. Whitelaw, A. Kehlenbeck, N. Petrovic, and L. Ungar. Web-scale named entity recognition. In *CIKM*, 2008.

[35] H. Zou and T. Hastie. Regularization and variable selection via the Elastic Net. *Journal of the Royal Statistical Society, Series B*, 67:301–320, 2005.

The Sweet Spot between Inverted Indices and Metric-Space Indexing for Top-K–List Similarity Search*

Evica Milchevski
University of Kaiserslautern
Kaiserslautern, Germany
milchevski@cs.uni-kl.de

Avishek Anand
L3S Research Center and
University of Hannover
Hannover, Germany
anand@l3s.de

Sebastian Michel
University of Kaiserslautern
Kaiserslautern, Germany
smichel@cs.uni-kl.de

ABSTRACT

We consider the problem of processing similarity queries over a set of top-k rankings where the query ranking and the similarity threshold are provided at query time. Spearman's Footrule distance is used to compute the similarity between rankings, considering how well rankings agree on the positions (ranks) of ranked items (i.e., the L1 distance). This setup allows the application of metric index structures such as M- or BK-trees and, alternatively, enables the use of traditional inverted indices for retrieving rankings that overlap (in items) with the query. Although both techniques are reasonable, they come with individual drawbacks for our specific problem. In this paper, we propose a hybrid indexing strategy, which blends inverted indices and metric space indexing, resulting in a structure that resembles both indexing methods with tunable emphasis on one or the other. To find the sweet spot, we propose an assumption-lean but highly accurate (empirically validated) cost model through theoretical analysis. We further present optimizations to the inverted index component, for early termination and minimizing bookkeeping. The performance of the proposed algorithms, hybrid variants, and competitors is studied in a comprehensive evaluation using real-world benchmark data consisting of Web-search–result rankings and entity rankings based on Wikipedia.

1. INTRODUCTION

One common way to counter the information deluge is the formation of concise rankings that allow users and algorithms to effectively and efficiently inspect the best performing items within a certain category. Ranking schemes are used to impose an order between items—such as Google's PageRank or more traditional OLAP-style aggregation and ranking functions used in databases for business intelligence and other forms of insight-seeking analyses. Besides tangible facts and objective ranking schemes, rankings are often also crowd-sourced through mining user polls on the Web, in portals such as IMDB (for movie ratings) or rankopedia.com, or specifically created by users in form of favorite lists on personal websites or used in dating portals for matchmaking. A core characteristic of such rankings is that they are rather tiny in length, compared to the global domain of items that could be ranked—consider the top-10 movies of all time compared to the total number of produced movies or the top-10 Web search results for Britney Spears compared to more than 100 million documents about her in Google's index. Access to rankings can serve ad-hoc information demands or give access to deeper analytical insights. Consider for instance the task of query suggestion in web search engines that is based on finding historic queries by their result lists with respect to the currently issued query, or dating portals that let users create favorite lists that are used to search for similarly minded mates.

As a generic access substrate for such services, we consider querying sets of top-k rankings by means of distance functions. That is, retrieving all rankings that have a distance to the query less than or equal to a user-provided threshold. We specifically focus on Spearman's Footrule that is the L1 distance metric between two rankings. Fagin et al. [18] show that there is a metric Spearman's Footrule adaptation for top-k rankings, whose ranked items do not necessarily match or overlap at all. Dealing with metrics immediately suggests employing metric data structures like M-trees [14] for indexing and similarity search. On the other hand, similar rankings, for reasonable query thresholds, should in fact overlap in some (or all) of the items they rank. Searching overlapping sets for ad-hoc queries [22, 30] or joins [25] is a well studied research topic. Inverted indices or signature trees are used to indexing tuples based on their set-valued attributes [22]. Such indices are very efficient to answer contained-in, equal-to, or overlaps-with queries, but do not exploit the distances between the indexed rankings as metric index structures do. In this work, we study a hybrid index structure that smoothly blends an inverted index with metric space indexing. With an assumption-lean but highly accurate theoretic cost model, we further show that the estimated sweet spot reaches runtime performances almost identical to the manually tuned one.

1.1 Problem Statement and Setup

As **input** we are provided with a set $\mathcal{T}$ of rankings τ_i (Table 1). Each ranking has a domain D_{τ_i} of items it contains. We consider fixed-length rankings of size k, i.e., $|D_{\tau_i}| = k$, but investigate the impact of various choices of k on the

*This work has been partially supported by the Excellence Cluster on Multimodal Computing and Interaction (MMCI) and the German Research Foundation (DFG) under grant MI 1794/1-1.

$\mathcal{T}$	
ranking id	**ranking content**
τ_1	$[2, 5, 4, 3]$
τ_2	$[1, 4, 5, 9]$
τ_3	$[0, 8, 5, 7]$

Table 1: Sample set $\mathcal{T}$ of rankings (items are represented by their ids).

query performance. The considered rankings do not contain any duplicate items.

Rankings are represented as arrays or lists of items, where the left-most position denotes the top ranked item. Without loss of generality, in the remainder of the paper, we assume that items are represented by their ids. The rank of an item i in a ranking τ is given as $\tau(i)$.

A distance function d quantifies the distance between two rankings—the larger the distance the less similar the rankings are. Therefore, for a given query ranking q, distance function d, and distance threshold θ, we want to find all rankings in $\mathcal{T}$ with distance below or equal to θ, that is,

$$\{\tau_i | \tau_i \in \mathcal{T} \wedge d(\tau_i, q) \leq \theta\}$$

In this work, we focus on the computation of Spearman's Footrule distance, but the proposed coarse index can be applied to any metric distance function. A more detailed introduction to rankings specifically top-k rankings, metric distance functions, and how to work with items i that are not in a ranking τ is described in Section 3.

The objective of this work is to study in-memory indexing and query processing techniques, with the overall aim to decrease the average query response time. We consider adhoc similarity queries over rankings, where the query ranking and query similarity threshold are specified at query time.

1.2 Contributions and Outline

In this work, we make the following contributions:

- we present a coarse index and a cost model that allows automated tuning of the coarsening threshold for optimal performance
- we derive distance bounds for early stopping / pruning inside position-augmented inverted indices—concepts that are largely orthogonal to each other and can be combined
- we show the results of a carefully conducted experimental evaluation involving a suite of algorithms and hybrids under realistic workloads derived from real-world rankings

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 presents background information on rankings and discusses distance functions for rankings. Section 4 introduces a coarse, hybrid index that indexes partitions of rankings. Section 5 describes a cost model that allows picking the sweet spot between inverted-index-access time and result-validation time. Section 6 shows how to compute distance bounds and to enable effective pruning of entire index lists at runtime. Section 7 presents the experimental evaluation, while Section 8 concludes the paper.

2. RELATED WORK

There is an ample work on computing relatedness between ranked lists of items, such as to mine correlations or anti-correlations between lists ranked by different attributes; like age and weight. Arguably, the two most prominent similarity measures are Kendall's tau and Spearman's Footrule. Fagin et al. [18] study comparing top-k lists, that is, lists capturing a subset of a global set of items, rendering the lists incomplete in nature. In the scenarios motivating our work, like similarity search of favorite/preference rankings, the lists are naturally incomplete, capturing, e.g., only the top-10 movies of all times. In this work, we focus on the computation of Spearman's Footrule distance, for which Fagin et al. [18] show that it retains its metric properties also for incomplete rankings under certain assumptions (cf., Section 3).

We primarily distinguish two indexing paradigms for handling ranked lists. First, considering the similarity metric among them and applying indexing techniques for metric spaces. Second, treating ranked lists as plain sets and indexing them using methods like inverted indices.

Helmer and Moerkotte [22] present a study on indexing set-valued attributes as they appear for instance in object-oriented databases. Retrieval is done based on the query's items; the result is a set of candidate rankings, for which the distance function can be computed. For metric spaces, data-agnostic structures for indexing objects are known, like the M-tree by Ciaccia et al. [14, 37]. For discrete metrics, the tree structure proposed by Burkhard and Keller [10] resembles an n-ary search tree, called BK-tree, where subtrees group items according to their (discrete) distance to the parent node. Similarly, Ganti et al. [21] present single-pass algorithms for clustering data in metric distance space using a R*-tree–style [7] structure for mapping objects to (evolving) clusters. The vantage-point tree [32, 36] partitions the space by choosing vantage points (pivots) that segment the space into two areas, similar to the k-d tree [8]. Chávez and Navarro [12] describe an algorithm to create non-overlapping partitions of data in a metric space based on pivots and fixed-diameter or fixed-size partitions; several ways to choose pivots are studied. We consider indexing clusters of rankings to shrink the size of the inverted index, by considering partitions of rankings within a pre-determined distance threshold (Section 4)—effectively trading-off cluster retrieval time and final result validation cost. The partitioning can be done in any of the above ways; we choose the BK-tree [10]. The book by Hanan Samet [26] gives a comprehensive overview of indexing techniques for metric spaces. The recent work by Wang et al. [34] propose MapReduce [16] algorithms for all-pairs similarity search in metric spaces. Previously, Jacox and Samet [24] proposed sequential algorithms for the similarity join problem in metric spaces.

Augmenting the inverted index with rank information allows computing the Footrule distance on the fly. For score-ordered index lists used in top-k query processing, there is a large variety of work. Most prominently, the family of threshold algorithms [18] and variants like the work by Bast et al. [4] that is emphasizing on disk-I/O optimal access. For k nearest neighbor (KNN) or similarity queries, the per-dimension information of the indexed objects is not pre-sorted by "score" as this depends on the query that is not known a priori. Work on KNN search in databases [9] transforms the KNN problem into a range query over the involved dimensions, that can be answered using standard database

τ	A ranking
$\tau(i)$	The rank of item i in ranking τ
$F(\tau_i, \tau_j)$	Footrule distance between τ_i and τ_j
$d(\tau_i, \tau_j)$	Distance between τ_i and τ_j
d_{max}	maximum distance between two rankings
$\mathcal{T}$	Set of rankings to be indexed
k	Size of rankings
$\mathcal{D}_\tau$	Items contained in ranking τ
$\mathcal{D}$	Global domain of items
q	Query ranking
θ	Similarity threshold, set at query time
θ_C	Maximum pairwise similarity within a partition
$\mathcal{P}_i$	A partition of rankings. Partitions are pairwise disjoint.

Table 2: Overview of notation used in this paper

indices that support range queries, like B+ trees [6]. Work on similarity join in databases [2, 27, 28] focuses on defining and implementing the similarity join as relational operators. Mamoulis [25] addresses processing joins between sets (relations) of tuples with set-valued attributes. Terrovitis et al. [29] considers containment queries over sets with skewed data distributions. The work in [30] proposes combination of trees and inverted files to answer superset, subset and equality queries over set-valued attributes. Recently, the most prominent technique for answering set similarity joins are the prefix-filtering based methods [35, 5, 11]. The main idea behind this method is to reduce the size of the inverted index. This is done by imposing a total ordering of the elements in the universe $\mathcal{U}$ (or what we refer to as the dictionary $\mathcal{D}$), sorting the elements in the records, and then, based on the threshold value, indexing only a prefix, and not the complete records. Similarly, we propose a technique for dropping some of the elements in the query (Section 6), however, our technique does not require the threshold value to be known during index construction. Additionally, we do not require a global ordering to be imposed on the items in the rankings, i.e., the rankings keep their original structure. Wang et al. [33] propose the AdaptJoin algorithm that improves on previous prefix filtering work by using variable length prefix scheme and a cost model that selects the most efficient prefix length for each object. They further propose the AdaptSearch algorithm for processing ad-hoc queries using the same adaptive framework. As rankings can also be seen as plain sets, AdaptSearch can be applied for computing relatedness between rankings as well.

When processing top-k, KNN, or similarity queries, the ultimate goal is to identify the final result objects as soon as possible, without exhaustive evaluation of scores/distances. In the NRA algorithm by Fagin et al. [19] over score-sorted index lists, this is achieved by maintaining score bounds for each seen object. We come back to this in Section 6.

3. BACKGROUND ON RANKINGS AND DISTANCE FUNCTIONS

Pairwise similar rankings can be retrieved by means of distance functions, like Kendall's Tau or Spearman's Footrule distance, over all pairs or selectively for a given query ranking. We first discuss metrics over complete rankings over a single domain and then we discuss results on computing distances for top-k lists (incomplete rankings).

Complete rankings are considered to be permutations over a fixed domain $\mathcal{D}$. We follow the notation by Fagin et al. [18] and references within. A permutation σ is a bijection from the domain $\mathcal{D} = \mathcal{D}_\sigma$ onto the set $[n] = \{1, \dots, n\}$. For a permutation σ, the value $\sigma(i)$ is interpreted as the rank of element i. An element i is said to be ahead of an element j in σ if $\sigma(i) < \sigma(j)$. For two permutations σ_1, σ_2 over the same domain, the Kendall's tau $K(\sigma_1, \sigma_2)$ and Spearman's Footrule $F(\sigma_1, \sigma_2)$ measures are two prominent ways to compute the distance between σ_1 and σ_2. Both measures are distance metrics, that is, they have symmetry property, i.e., $d(x, y) = d(y, x)$, are regular, i.e., $d(x, y) = 0$ iff $x = y$, and suffice the triangle inequality $d(x, z) \leq d(x, y) + d(y, z)$, for all x, y, z in the domain. Spearman's Footrule metric is the L_1 distance between two permutations, i.e., $F(\sigma_1, \sigma_2) = \sum_i |\sigma_1(i) - \sigma_2(i)|$ and in this work we specifically focus on this metric, but the proposed coarse index can be applied to any metric distance function. We refer the reader to Table 2 for an overview of the notation used in this paper.

We consider incomplete rankings, called top-k lists in [18]. Formally, a top-k list τ is a bijection from D_τ onto $[k]$. The key point is that individual top-k lists, say τ_1 and τ_2 do not necessarily share the same domain, i.e., $D_{\tau_1} \neq D_{\tau_2}$. Fagin et al. [18] discuss how the above two measures can be computed over top-k lists.

There exists a Spearman's Footrule adaptation that is also a metric for top-k lists by considering an artificial rank l for items not contained in a ranking, i.e., $\tau(i) = l$ if $i \notin D_\tau$. Consider the rankings $\tau_1 = [2, 5, 6, 4, 1], \tau_2 = [1, 4, 5]$, and $\tau_3 = [0, 8, 4, 5, 7]$. For a rank $l = 6$ for not-contained items, we obtain $F(\tau_1, \tau_2) = 15$, $F(\tau_2, \tau_3) = 17$, and $F(\tau_1, \tau_3) = 22$.

In this work, we assume that $\tau(i)$ takes values from 0 to $k - 1$ (instead of 1 to k), and we fix the value of l to k as suggested in [18]. It is clear that this does not affect our algorithms. We further consider only rankings of same size k, thus the largest possible value of the Footrule distance is $k \times (k + 1)$ and occurs if two disjoint rankings are compared. The smallest distance is 0, for the compared rankings are identical. In the rest of the paper, for ease of presentation, we use normalized values for the Footrule distance and θ, ranging from 0 to 1, i.e., $d_{max} = 1$.

4. FRAMEWORK

Rankings can be considered as plain sets and accordingly indexed in traditional inverted indices [22] that keep for each item a list of rankings in which the item appears. At query time such a structure allows efficiently finding those rankings that have one or more items in common with the query ranking. A compact example is given below:

$$\text{item a} \longrightarrow <\tau_1, \tau_5, \tau_7> \qquad \textit{inverted index}$$
$$\text{item b} \longrightarrow <\tau_4, \tau_9, \tau_{12}, \tau_{19}>$$

The key point of using inverted indices is their ability to efficiently reduce the global amount of all rankings to potential candidates by eliminating the rankings with maximum distance d_{max} to the query. This is done in the first query processing phase, namely the **filtering phase**. In this phase, for a given query ranking q and a user defined threshold θ, the inverted index is queried for each item in $\mathcal{D}_q$. The obtained index lists are merged to identify all rankings that have at least one overlapping item with the query ranking q. These are considered candidates.

For each of them, the distance function $d(q, \tau)$ is evaluated to identify the true results, i.e., the rankings where $d(q, \tau) \leq \theta$. This is done in the **validation phase**. We refer to this as the Filter and Validate (F&V) algorithm. Naturally, we assume that the query threshold θ is strictly smaller than the maximum possible distance d_{max}.

Although the inverted index is good for finding rankings (sets) that intersect with the query, the F&V comes with two drawbacks:

(i) It naively indexes all rankings and, hence, is of massive size, despite the fact that often rankings are (near) duplicates

(ii) The validate phase evaluates the distance function on each ranking separately, although known metric index structures suggest pre-computing distances among (similar) rankings for faster identification of true results

While directly using metric index structures, like M-Trees [14] or BK-Trees [10], appears promising at first glance, they are not ideal for boiling down the space to intersecting rankings. In fact, we show in our experiments that using metric data structures is an order of magnitude slower than using pure inverted indexes.

To harness the pruning power of inverted indices but at the same time not to ignore the metric property of the Footrule distance, we present a hybrid approach that blends both performance sweet spots by representing near duplicate rankings by one representative ranking, which is then put into an inverted index. That way, depending on how aggressive this coarsening is, the inverted index drastically shrinks in size, hence, lower response time, and the validation step is benefiting from the fact that near duplicate rankings are represented by a metric index structure.

Below, we describe more formally how such an index organisation is realized and how queries are processed on top of it. We present a highly accurate cost model that allows trading-off the coarsening threshold to find the optimal trade-off between the inverted index cost and the cost to validate rankings in the metric index structure.

4.1 Index Creation

The aim is to group together rankings that are similar to each other—with a quantifiable bound on the maximum distance. That is, partitions $\mathcal{P}_i$ of similar rankings are created, and each represented by one $\tau_m \in \mathcal{P}_i$, the so called medoid of the partition. It is guaranteed that $\forall \tau_i \in \mathcal{P} : d(\tau_m, \tau) \leq \theta_C$. The distance bound θ_C is called the partitioning threshold. We write $\tau_m \prec \tau$ to denote that ranking τ is represented by ranking (medoid) τ_m.

To find partitions of rankings, we employ a BK-tree [10], an index structure for discrete metrics, such as the Footrule distance. Figure 1 depicts the general shape of such a BK-tree. Ignoring for a moment the different colors and black, solid circles: each node represents an object (here, ranking) and maintains pointers to subtrees whose root has a specific, discrete distance. We create such a BK-tree for the given rankings. Then, in order to create partitions of similar rankings, the tree is traversed and, for each node, the children with distance above θ_C are considered in different partitions. The procedure continues recursively on these children. The children within distance $\leq \theta_C$ are forming a partition with their root node, which acts as the medoid. In Figure 1, each partition is illustrated by its root (representative ranking) shown as a black, solid circle, and the green subtrees below it (those with distance 1 or 2). A partition is not represented as a plain set (or list) of rankings, but by the corresponding subtree of the BK-tree. The immediate benefit is that these subtrees (that are full-fledged BK-tree themselves) are used to process the original query (with threshold θ) on the clusters, without the need to perform an exhaustive evaluation of the partition's rankings. Alternatively, any algorithm that creates (disjoint) partitions of objects within a fixed distance bound can be used, such as the approach by Chávez and Navarro [12], which randomly picks medoids, assigns objects to medoids, and continues this procedure until no object is left unassigned. We use this simple model to reason about the trade-offs of our algorithm below.

Irrespective of the way to find medoids and their partitions, medoids are rankings, too, and can be indexed using inverted indices. In Section 6 we further propose techniques for more efficient retrieval of the rankings indexed with an inverted index.

4.2 Query Processing

LEMMA 1. *For given query threshold θ and partitioning threshold θ_C, at query time, for query ranking q, all medoids τ_m with distance $d(\tau_m, q) \leq \theta + \theta_C$ need to be retrieved in order not to miss a potential result ranking.*

Lemma 1 ensures that rankings $\{\tau_i | \tau_m \prec \tau_i \wedge d(\tau_i, q) \leq \theta \wedge d(\tau_m, q) > \theta\}$ will not be omitted from the result set. In other words, Lemma 1 avoids missing result rankings with distance $\leq \theta$, which are represented by a medoid with distance $> \theta$. On the other hand, since the medoids are indexed using an inverted index, we assume that $\theta + \theta_C < 1$. This is needed because medoids τ_m that are not overlapping with q at all, cannot be retrieved from the inverted index.

For each of the found medoids τ_m (i.e., $d(\tau_m, q) \leq \theta + \theta_C$), the rankings $\mathcal{R} := \{\tau | \tau_m \prec \tau\}$ are potential result rankings. For each such candidate ranking $\tau_i \in \mathcal{R}$ it needs to be checked if in fact $d(q, \tau_i) \leq \theta$. The rankings $\tau_i \in \mathcal{R}$ with $d(q, \tau_i) > \theta$ are so called *false positives* and according to Lemma 1 there are *no false negatives.* As for each affected medoid τ_m, the rankings in $\mathcal{R}$ are represented in form of a BK-tree (or any other metric index structure), it is the task of this tree to identify the true result rankings (i.e., eliminating the false positives).

Algorithm 1, depicts the querying using the relaxed query threshold, and the subsequent retrieval of result rankings. In this algorithm, as well as in the actual implementation, the partitions, represented by the medoids, are arranged as BK-trees, created at partitioning time.

It is clear that the partitioning threshold θ_C affects the cost for querying the metric index structure: The larger the partitions are (i.e., the larger θ_C is) the larger is the tree to be queried. On the other hand, then, there are less medoids to be indexed in the inverted index. This apparent tradeoff is theoretically investigated in the following section to find the design sweet spot between the naive inverted index and the case of indexing the entire set of rankings in one metric index structure.

```
method: processCoarse
input: QueryProcessor over Medoids qp, double $\theta$, $\theta_C$,
       Map:Int→ BK-Tree map
output: list of query results rlist
1  rTemp ← qp.execute($\theta$+$\theta_C$) ▷ query with relaxed threshold
2  for each id ∈ rTemp
3      tree ← map[id]
4      rList.addAll(tree.execute($\theta$))
5  return rList
```

Algorithm 1: Query processing using the coarse index.

5. PARAMETER TUNING

Setting the clustering threshold θ_C allows tuning the performance of the coarse index. For a clustering threshold $\theta_C = 0$, only duplicate rankings are grouped together, whereas for $\theta_C = 1$ there is only one large group that consists of all rankings. That means, for larger θ_C the inverted index becomes smaller, with more work to be done at validation time inside the retrieved clusters. For smaller θ_C the inverted index is larger, but clusters are smaller, hence, less work to be done in the validation phase. There are, hence, two separate costs: **filtering cost**—the cost for querying the inverted index, and, **validation cost**—the cost for validating the partitions represented by the medoids returned as results by the inverted index, in order to get the final query answers.

We try to make as few assumptions as possible and for now we assume we know only the distribution of pairwise distances. That is, for a random variable X that represents the distance between two rankings, we know the cumulative distribution function $P[X \leq x]$, hence, we know how many rankings of a population of n rankings are expected to be within a distance radius r of any ranking, i.e., $n \times P[X \leq \theta_C]$. We assume that medoids are also just rankings (by design) and are accordingly distributed. According to the clustering method described by Chávez and Navarro [12], we randomly select medoids, one after the other. After each selected medoid, all rankings that are not yet assigned to any medoid before and that are within distance θ_C to the current medoid are assigned to it. The process ends as soon as no ranking is left unassigned:

The radius r of the created partitions around the medoids is modeled as $P[X \leq \theta_C]$. We are interested in the number of medoids that need to be created to capture all rankings in the database. This resembles the *coupon collector problem* [20]. The solution to this problem describes how many coupons a collector needs to buy, in expectation, to capture all distinct coupons available. The first acquired coupon is unique with probability 1. The second pick is not seen before with probability $(c-1)/c$; c denoting the total number of distinct coupons. The third pick with probability $(c-2)/c$, and so on. In the case of medoids and their partitions, we specifically consider the variant of the coupon collector problem with package size larger or equal to one, i.e., batches of coupons are acquired together. Within each such package, there are no duplicate coupons. Figure 2 depicts the generic sampling of the ranking space, where fixed-diameter circles are forming the partitions around the medoid at the center. The deviation from the standard coupon collectors problem is that for picking medoids, in each round of picks, the medoid itself has not been selected before. Thus, the number

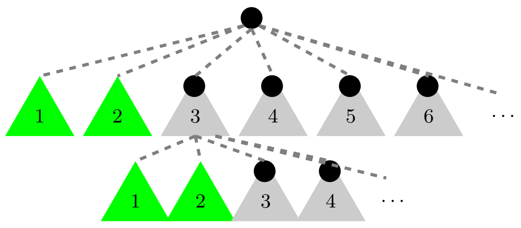


Figure 1: Creating partitions based on the BK-tree. The green (distance 1 and 2) subtrees are indexed by their parent node (medoid, as black dot). Distance 0 is not shown here.

of "coupons" that need to be acquired to get the i^{th} distinct coupon, given package size $p = P[X \leq \theta_C] \times n$, and a total of c distinct coupons, which in our case is the number of distinct rankings n is then:

$$h(n,i,p) = \begin{cases} 1, & \text{if } i \,\mathbf{mod}\, p = 0 \\ \frac{n-(i \,\mathbf{mod}\, p)}{n-i}, & \text{otherwise} \end{cases} \quad (1)$$

And overall, the number of medoids (packages) is given as

$$M(n,\theta_C) = p^{-1} \sum_{i=0}^{n-1} h(n,i,p) \quad (2)$$

This gives us the expected number of medoids indexed by the inverted index. Next, we first reason about the cost for validating the partitions, and then we discuss the filtering cost, i.e., the cost for querying the inverted index.

Cost for Validating Partitions

The number of medoids retrieved is following again the given distribution of pairwise distances. Since we query the inverted index with threshold $\theta + \theta_C$ we obtain

$$E[\textit{retrieved medoids}] = P[X \leq \theta + \theta_C] \times M \quad (3)$$

where M, for brevity, denotes $M(n, \theta_C)$.

Assuming that the retrieved medoids have the same size on average, i.e., n/M for a total number of rankings n, we have

$$E[\textit{candidate rankings}] = P[X \leq \theta + \theta_C] \times n \quad (4)$$

candidate rankings retrieved that need to be checked against the distance to the query ranking. This is also very intuitive.

For the case of brute-force evaluation of such candidate rankings this is multiplied with the cost of computing the distance measure. The cost of representing the partitions by full-fledged BK-tree is expected to be lower, but it introduces a complexity to the model. Our goal is to provide an easy to compute, and yet accurate model. For a more complex reasoning about the cost of querying the BK-tree we refer the reader to [3].

Cost for Retrieving Partitions

When querying the inverted index with a threshold $\theta + \theta_C$ to find the resulting medoids, the overall cost is based on the average index list length and the final medoids to be checked against the threshold. We should first estimate the average size of an index list in an inverted index.

We assume that the popularity of items in the rankings follows Zipf's law with parameter s. Sorting all items by their popularity (frequency of appearance in the rankings),

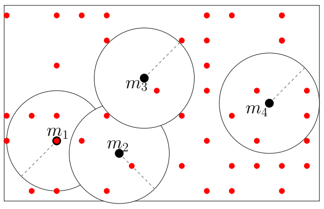


Figure 2: Four medoids with fixed-diameter partitions.

the law states that the frequency of the item at rank i is given by $f(i; s, v) = \frac{1}{i^s H_{v,s}}$, where $H_{v,s}$ is the generalized harmonic number and v is the total number of items. The size of the index list for an item is equal to the number of rankings that contain the item, i.e., $n \times f(i; s, v)$ for the i^{th} most popular item; where n is the number of indexed rankings. Consider a random variable Y representing sizes y_i of index lists for items i. We are interested in $E[Y] = \sum_i y_i\, P[y_i]$ and assume that the chance of item i, that is the i^{th} most popular item, to be selected as a query item is following the same Zipf distribution, $f(k; s, v)$. That means, the items appearing frequently in the data are also used often in the queries. The average size of an index list is then given as $E[Y] = \sum_i n \times f(i; s, v)^2$. This is a generic result for inverted indices, which in our cost model is applied on an inverted index over M medoids (not n rankings) that together have v' distinct items; v' is derived thereafter, so the expected length of an inverted list for the inverted index is:

$$E[\mathit{index\ list\ length}] = \sum_i M \times f(i; s, v')^2 \tag{5}$$

For each query, k such index lists need to be accessed. This is one part of the cost caused by the retrieval of the medoids. For these $k \times E[\mathit{index\ list\ length}]$ medoids, we have to compute the distance function, assuming that there are no duplicate medoids retrieved.

The expectation of distinct items v' within the medoids is derived as follows. The probability that an item, out of a global domain of v items, is not selected into a single ranking of size k is $(\frac{v-1}{v})^k$, but we do know that a ranking does not contain duplicate items, hence, $P[\neg \mathit{selected}] = \frac{v-1}{v} \times \frac{v-2}{v-1} \cdots \frac{v-k}{v-k+1} = \Pi_{i=0}^{k} \frac{v-i}{v-i+1} = 1 - (\frac{k}{v})$. The probability that an item, out of a global domain of v items, is not selected into a single ranking of size k, knowing that the items in the ranking are unique, is $P[\neg \mathit{selected}] = 1 - (\frac{k}{v})$. The probability *not* to be selected in *any* of the M medoid rankings is then $(1 - \frac{k}{v})^M$. And thus

$$E[v'] = v \times \left(1 - \left(1 - \frac{k}{v}\right)^M\right) \tag{6}$$

To compute the overall cost, the above estimates are combined as shown in Table 3. To bring both parts of the overall cost to a comparable unit, we precompute the cost (runtime) of a single Footrule computation $Cost_{Footrule}(k)$ (for various k) and the cost (runtime) to merge k lists of a certain size, $Cost_{merge}(k, size)$.

Figure 3 shows the model for vary θ_C for the two datasets used in the experimental evaluation (we refer the reader to Section 7 for a description of the datasets). We empirically estimated the skewness parameter s from samples of the datasets—$s = 0.87$ for the New York Times dataset (left plot) and $s = 0.53$ for the Yago dataset (right plot)—and fitted it in the above estimate of the expected index list length.

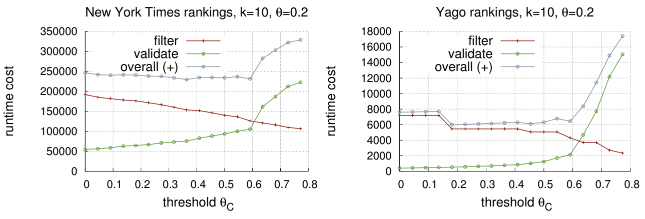


Figure 3: The behavior of the theoretically derived performance for varying θ_C.

Find medoids for query:	
Inv. Index Cost:	$Cost_{merge}(k, \sum_i f(i; s, v')^2 \times M)$
	$+$
Validation Cost:	$k\left(\sum_i f(i; s, v')^2 \times M\right) \times Cost_{Footrule}(k)$
Validation of retrieved rankings:	
Validation Cost:	$n \times P[X \leq \theta + \theta_C] \times Cost_{Footrule}(k)$

Table 3: Model of query performance (∼runtime) of the coarse index.

6. INV. INDEX ACCESS & OPTIMIZATIONS

Medoids are rankings as well and thus they can be indexed using inverted indices. In this section, two optimizations over inverted indices are presented.

First, a minimum-overlap criterion is derived; it indicates how many of the k index lists can be dropped from consideration, guaranteeing that no true result ranking can possibly be missed.

For the second optimization, for a query ranking of size k, the k corresponding index lists are accessed one after the other, and the contained information in each list in the form of $(\tau_i, \tau(i))$ are continuously aggregated for each (seen) ranking. For each ranking observed during accessing the index lists, upper and lower bounds for the true distance are derived, to allow accepting or rejecting final result rankings early.

6.1 Pruning by Query-Ranking Overlap

Consider a ranking τ with $\mathcal{D}_\tau \cap \mathcal{D}_q = \emptyset$, i.e., the items in τ are not at all overlapping with the query's items. It is easy to see that the Footrule distance is $F(\tau, q) = k \times (k+1) = L(k)$, considering rankings of length k. $L(k)$ is used to denote this lowest possible distance[1]. In the case of zero overlap, $L(k)$ is also the exact distance. In general, considering an overlap of size ω between $\mathcal{D}_q$ and $\mathcal{D}_\tau$, the smallest possible Footrule distance $L(k, \omega)$ in that case is given when the ω overlapping items are perfectly matched and positioned in the top of both lists, hence, $L(k, \omega) = L(k - \omega)$. For a given query threshold θ, rankings with an overlap of ω items can be safely ignored if $L(k, \omega) > \theta$. In practice, this means that some index lists can be entirely omitted from being accessed.

[1]We use the naming lower and upper bounds for distances instead of best and worst distances, for clarity.

It is immediately clear how to turn this insight into enhancements of algorithms that work with an inverted index: Solving $L(k, \omega) = \theta$ for ω tells that rankings τ with $F(\tau, q) \leq \theta$ must not have an overlap smaller than $\omega = \lfloor 0.5(1 + 2k - \sqrt{1 + 4\theta}) \rfloor$. From this it immediately follows that $k - \omega + 1$ index lists are sufficient to retrieve all the candidate lists since a ranking missing from these lists must have an overlap smaller than ω.

If we further take into consideration the position of the ω overlapping items, i.e., that they are positioned at the top of both lists, then we can ensure correctness by retrieving $k-\omega$ lists if at least one of the retrieved lists is of an item positioned in the top ω places. In this case, we can miss rankings that have an overlap of ω with the query, but we will never miss rankings that have overlap of ω where these ω items are positioned at the top ω places. This leads immediately to the following lemma.

LEMMA 2. *For given query threshold θ and ranking size k, $k - \omega$ index lists are sufficient to retrieve all the candidate rankings τ with $F(\tau, q) \leq \theta$, where $\omega = \lfloor 0.5(1 + 2k - \sqrt{1 + 4\theta}) \rfloor$.*

This is a generic result, independent of the actual choice of the index lists that can be dropped. Still, the expected impact of the candidate pruning is larger if the largest lists are dropped. In fact, experiments will show that specifically for the query-log–based benchmark, drastic performance gains can be enjoyed, literally for free. For the remaining lists and rankings within, the exact distance still needs to be determined, as there are obviously so called *false positives* with distance larger than the query threshold. But the above lemma guarantees that there are *no false negatives*, i.e., no ranking τ with $F(\tau, q) \leq \theta$ is missed.

Algorithms that make use of this dropping of entire index lists carry the suffix **+Drop** in the title.

6.2 Partial Information

Instead of having only ranking ids stored in the inverted index, such that an additional lookup is required to get the actual ranking content, we can augment the inverted index to make it hold the rank information as well, such that the true distance can be directly computed.

inverted index w/ ranks

item a $\longrightarrow$ <(τ_1 : 3),(τ_5 : 1),(τ_7 : 4)>

item b $\longrightarrow$ <(τ_4 : 2),(τ_9 : 11),(τ_{12} : 1),(τ_{19} : 2)>

In a **List-at-a-Time** fashion, the individual index lists determined by the query are accessed one after the other. Similarly to the NRA algorithm by Fagin et al. [19], for a ranking τ that has been seen only in a subset of the index lists, we can compute bounds for its final distance. This is done by keeping track of the common elements seen between the query q and ranking τ. The lower and upper bounds are computed by reasoning about the yet unseen elements: A lower bound distance $L(\tau, q)$ is given by assuming the best configuration of the unseen elements, that is, the remaining elements are common to both q and τ, and are additionally present in the same ranks in both rankings. Thus, their partial contribution to the Footrule distance is zero.

The upper bound distance $U(\tau, q)$ is obtained when none of the (yet) unseen elements in τ will be present in the query q. The partial distance contribution of such an item i, at rank $\tau(i)$ in τ is $|k - \tau(i)|$, and overall we have

$\tau_0 = [1, 2, 3, 4, 5]$	$\tau_5 = [4, 5, 1, 2, 3]$
$\tau_1 = [1, 2, 9, 8, 3]$	$\tau_6 = [1, 6, 2, 3, 7]$
$\tau_2 = [9, 8, 1, 2, 4]$	$\tau_7 = [7, 1, 6, 5, 2]$
$\tau_3 = [7, 1, 9, 4, 5]$	$\tau_8 = [2, 5, 9, 8, 1]$
$\tau_4 = [6, 1, 5, 2, 3]$	$\tau_9 = [6, 3, 2, 1, 4]$

Table 4: Sample set $\mathcal{T}$ of rankings

$$U(\tau, q) = L(\tau, q) + \sum_{i\, unseen} |k - \tau(i)|$$

The bounds allow pruning of candidates: If $L(\tau, q) > \theta$ we know that τ is not a result ranking, since $L(\tau, q)$ is monotonically non-decreasing. Similarly, if $U(\tau, q) \leq \theta$, we report τ as the result, as $U(\tau, q)$ is monotonically non-increasing. For small values of θ, many candidates can be evicted early on in the execution phase. For larger values of θ, candidate results can be reported early—reducing bookkeeping costs.

Consider for instance the set $\mathcal{T}$ of the rankings presented in Table 4 and a query $q = [7, 6, 3, 9, 5]$. The index list for item 7 is:

item 7 $\longrightarrow$ <(τ_3 : 0),(τ_6 : 4),(τ_7 : 0)>

We can compute the bounds for the seen rankings, τ_3, τ_6, and τ_7. For all these rankings, we know the seen element is item 7 and we have 4 unseen elements, since $k = 5$. Thus, $L(\tau_3, q) = L(\tau_7, q) = 0$ and $L(\tau_6, q) = 4$, as $\tau_3(7) - q(7) = \tau_7(7) - q(7) = 0$, and $\tau_6(7) - q(7) = 4$ and for the unseen items we assume they are on the same position in all rankings. $U(\tau_3, q) = U(\tau_7, q) = 20$ and $U(\tau_6, q) = 24$, as we assume that all of the unseen elements are not present in τ_3, τ_6, and τ_7.

These distance bounds are used in the following online aggregation algorithm that encounters partial information. Algorithms that make use of this pruning for partial information carry the suffix **+Prune** in their title.

6.3 Blocked Access on Index Lists

When index lists are ordered according to the rank values, since the ranks are integers, there might be a sequence of index lists whose ranks are the same. We refer to this sequence of index lists as a block of index lists. Formally, we let the block $\mathcal{B}_{i@j}$ to denote the set of rankings in which item i appears at position j. We additionally have a secondary index, one for each index list, which stores the offsets of the individual blocks.

The advantage with such an index list organization strategy is that processing the entire index list can be avoided in many cases. We describe this in detail. It is obvious that result candidates which have a partial distance greater than θ can be pruned out. In such an index organization approach, we avoid processing blocks which would produce candidates with a partial distance greater than θ. Given a query $q = [q_1, \ldots, q_k]$ with a threshold θ, all result candidates obtained while traversing the block $\mathcal{B}_{i@j}$ have a partial distance of at least $|j - i|$. Thus, we modify the List-at-a-Time algorithm so that blocks, $\mathcal{B}_{i@j}$, where $|j - i| > \theta$ are omitted, avoiding processing the bulk of the index list.

Consider for instance the inverted index in Figure 4, constructed according to the rankings in Table 4. For the query $q = [3, 2, 1]$ and $\theta = 1$, blocks $\mathcal{B}_{3,1}$ need to be accessed for item 3, $\mathcal{B}_{2,1}$, $\mathcal{B}_{2,1}$ and $\mathcal{B}_{2,3}$ for item 2. Finally, blocks $\mathcal{B}_{1,2}$, $\mathcal{B}_{1,3}$ and $\mathcal{B}_{1,4}$ for item 1. In the process 17 out of 28 index item 1$\rightarrow$ $\boxed{(\tau_0:0), (\tau_1:0), (\tau_6:0)}, \boxed{(\tau_3:1), (\tau_4:1), (\tau_7:1), (\tau_{10}:1)}, \boxed{(\tau_2:2), (\tau_5:2)}, \boxed{(\tau_9:3)}, \boxed{(\tau_8:4)}$

item 2$\rightarrow$ $\boxed{(\tau_8:0)}, \boxed{(\tau_0:1), (\tau_1:1)}, \boxed{(\tau_6:2), (\tau_9:2)}, \boxed{(\tau_2:3), (\tau_4:3), (\tau_5:3), (\tau_{10}:3)}, \boxed{(\tau_7:4)}$

item 3$\rightarrow$ $\boxed{(\tau_9:1)}, \boxed{(\tau_0:2)}, \boxed{(\tau_6:3)}, \boxed{(\tau_1:4), (\tau_4:4), (\tau_5:4)}$

item 4$\rightarrow$ $\boxed{(\tau_5:0)}, \boxed{(\tau_{10}:2)}, \boxed{(\tau_0:3), (\tau_3:3)}, \boxed{(\tau_2:4), (\tau_9:4)}$

. . .

Figure 4: Inverted Index for rankings in Table 4 with highlighted blocks of same-rank entries

lists are processed which accounts for less than 50% index lists being accessed.

7. EXPERIMENTS

We implemented the described algorithms in Java 1.7 and report on the setup and results of an experimental study. The experiments are conducted on a quad-core Intel Xeon W3520 @ 2.67GHz machine (256KiB, 1MiB, 8MiB for L1, L2, L3 cache, respectively) with 24GB DDR3 1066 MHz (0.9 ns) main memory.

Datasets

Yago Entity Rankings: We have mined top-k entity rankings out of the Yago knowledge base, as described in [23]. The facts, in form of subject/predicate/object triples, are used to define constraints, for which the qualifying entities are ranked according to certain criteria. For instance, we generate rankings by focusing on type building and predicate located in New York, ranked by height. This dataset, in total, has 25,000 rankings.

NYT: We executed 1 million keyword queries, randomly selected out of a published query log of a large US Internet provider, against the New York Times archive [31] using a unigram language model with Dirichlet smoothing as a scoring model. Each query together with the resulting documents represents one ranking.

The two datasets are naturally very different: while the Yago dataset features real world entities that each occur in few rankings, the NYT dataset has many popular documents that appear in many query-result rankings.

Algorithms under Investigation

- the baseline approaches Filter and Validate (**F&V**) and Merge of Id-Sorted Lists (**ListMerge**) both described below
- filter and validate technique combined with the optimization based on dropping entire index lists (**F&V+ Drop**)
- blocked access with pruning (**Blocked+Prune**)
- blocked access with pruning based on both overlap and pruning (**Blocked+Prune+Drop**)
- query processing on the coarse index using the F&V technique (**Coarse**)
- query processing on the coarse index using the F&V+ Drop technique (**Coarse+Drop**)
- a competitor **AdaptSearch**, and **Minimal F&V** algorithm, both described below

Next to the actual algorithms, we implemented a minimal Filter and Validate algorithm (**Minimal F&V**) that has for each query materialized a single index list in an inverted index that contains exactly the true query-result rankings. For each of these, the Footrule distance is computed. The cost for the single index lookup and the Footrule computations serves as a lower bound for the performances of the discussed algorithms.

We also implemented **AdaptSearch** [33] as the most recent and competitive work on ad-hoc set similarity search in main memory. We implemented AdaptSearch by following the C++ implementation of the AdaptJoin algorithm available online[2]. We computed the size of the prefix of the query using the overlap threshold ω derived in Section 6. In the validation phase, AdaptSearch computes the Footrule distance for each of the candidate rankings.

The implementation of the M-tree is obtained from [15]. We implemented the BK-tree ourselves, according to the original work in [10]. The inverted index implementations make use of the Trove library[3].

Merge of Id-Sorted Lists with Aggregation: If the information within each index list is sorted by ranking id, and further contains rank information, the problem of computing the actual distances of the rankings to the query ranking can be achieved using a classical merge "join" of id-sorted lists. This is very efficient, in particular as the index lists do not contain any duplicates. Cursors are opened to each of the lists, and the distances of each ranking is finalized on the fly. There is no bookkeeping required as, at any time, only one ranking is under investigation (the one with the lowest id, if sorted in increasing order). Rankings do either qualify the query threshold or not. It is clear that this algorithm is threshold-agnostic, that is, its performance is not influenced by the query threshold θ; the index lists have to be read entirely.

We mainly focus on rankings of size 10 since in a previous study [1] we observed that at ranker.com most common are rankings of size 10.

Performance Measures

- Wallclock time: For all algorithms we measure the wallclock time needed for processing 1000 queries.
- Distance function calls: For the filter&validate algorithms, specifically F&V, F&V+Drop, Blocked+Prune+Drop, Coarse, and Coarse+Drop, we measure the number of distance function computations performed.

For the coarse index processing techniques, we also investigate the performance of the individual phases.

7.1 Query Processing Performance

Inverted Index vs. Metric Index Structures

We first compare the two main concepts of processing similarity queries over top-k rankings: First, the use of met-

[2]https://github.com/sunlight07/similarityjoin
[3]http://trove.starlight-systems.com/

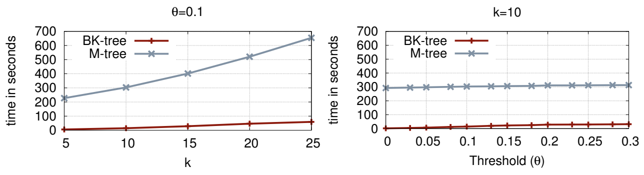


Figure 5: Performance of the M-tree vs. BK-tree (NYT)

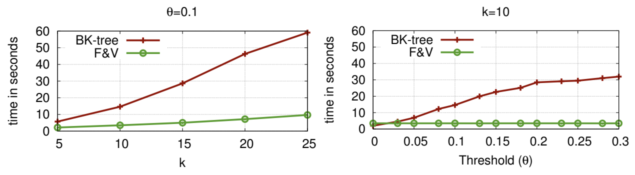


Figure 6: Performance of the BK-tree vs. the performance of inverted index (NYT)

ric index structures is compared, here, represented by the BK-tree and the M-tree [37] (Figure 5). Second, the use of inverted indices is compared to the BK-tree (Figure 6).

Figure 5 reports the query performance of the BK-tree compared to the M-tree and Figure 6 on the query performance of the BK-tree index structure versus the plain query processing using the inverted index with subsequent validation, i.e., filter and validate, F&V. We see that the inverted index performs orders of magnitudes better than the M-tree. Although the M-tree is a balanced index structure it still performs worse than the BK-tree. Chávez et al. [13] show that balanced index structures perform worse than unbalanced ones in high dimensions—we calculated the intrinsic dimensionality of both datasets to be around 13 (cf. [13] for the definition of intrinsic dimensionality). Despite the better performance of the BK-tree, the inverted index still outperforms it. Hence, only techniques using the inverted index paradigm are further studied.

Coarse Index Performance Based on θ_C

Next, we studied the performance of the coarse index for different θ_C values. We focus on the performance of the coarse index combined with the F&V technique as this combination resembles the model presented in Section 4 most. In Figure 7, the filtering and validation times are shown when varying θ_C and fixed $k = 10$, for both datasets. We see that the curves resemble the ones plotted for the cost model in Figure 3. Both dataset show a similar behavior of the execution time. The filtering time is reducing as we increase the value of θ_C, since the number of indexed medoids reduces. The validation time, on the other hand, is rising, since the size of the partitions is increasing proportionally with θ_C. Most importantly, we see that we can find a specific value of θ_C for which the coarse index performs optimally and this value depends on the value of $\theta + \theta_C$ as modeled in Section 4.

Cost Model Correctness

The performance of the coarse index if the trade-off value of θ_c as computed by the model is chosen, is shown in the plots in Figure 7 as a small rectangle. The vertical line denotes the difference between the performance of the coarse index in case of the two trade-off θ_c values—the modeled optimal one and the real optimal one. We observe that except for $\theta = 0.1$, for the NYT dataset, the difference in performance

	$\theta = 0.1$	$\theta = 0.2$	$\theta = 0.3$
NYT	29.47	10.23	4.75
Yago	3.28	0.41	2.38

Table 5: Difference in ms between the minimal performance of the coarse index, and the performance for the theoretically computed best value of θ_c ($k = 10$)

is smaller than 11ms (Table 5). For $\theta = 0.1$ the difference is 29.47ms. For the Yago dataset, the difference in performance is less than 4ms for any value of θ.

As we are considering the task of processing ad-hoc queries, even choosing the optimal value of θ_C for some previously defined maximum value of θ would result in a performance close to the optimal one, as the performance of the coarse index remains stable in this region. The major increase in the performance happens for very small values of θ_C or larger than the optimal θ_C. We show this in the experiments comparing different algorithms, where we set $\theta_C = 0.5$—the optimal value for $\theta = 0.3$.

We also measure the performance of the coarse index combined with the F&V+Drop technique as this should result in even bigger performance gains. For this technique, we measured the optimal value for θ_C to be 0.06, since for smaller values of $\theta + \theta_C$ we can drop more index lists.

Comparison of Different Algorithms

Next, we study the performance of different query processing methods performed over the two datasets; for rankings of size 10 and 20 and θ ranging from 0 to 0.3. First, in Figure 8 we compare the performance of the coarse index with the remaining techniques, for the NYT dataset. For a better visibility, we group the algorithms in the plots in two groups. The first (left) group contains the Coarse and Coarse+Drop techniques, the two baseline approaches F&V and ListMerge, and the competitors AdaptSearch and Minimal F&V. The second (right) group contains the remaining hybrid techniques.

We see that for all threshold values the coarse index, with and without dropping index list, significantly outperforms the AdaptSearch algorithm. In fact, the Coarse+Drop index outperforms the competitor by at least factor of 34. The coarse index outperforms the Minimal F&V technique by a factor of up to 7, since the number of Footrule distance function calls reduces significantly as shown in Figure 10. Dropping entire lists from the query even further boosts the performance of the coarse index, and results in up to 24 times better performance than the Minimal F&V. The baseline approaches, although threshold agnostic, perform worse than the rest of the algorithms. Increasing the values of θ degrades the performance of all the processing techniques except for the baseline F&V and ListMerge techniques, as they are threshold agnostic. In fact, because of its simple and efficient implementation, the ListMerge even outperforms the AdaptSearch algorithm for $\theta \geq 0.1$ for rankings with $k = 10$. For $k = 20$, since we increase the number of lists that need to be merged, the performance of the ListMerge is worse and thus the AdaptSearch outperforms it for all values of θ.

For rankings of size 10, all hybrid techniques outperform AdaptSearch, but not the coarse index. The Blocked+Prune algorithm dynamically computes the best score for the yet unseen blocks to decide when to terminate further schedul-

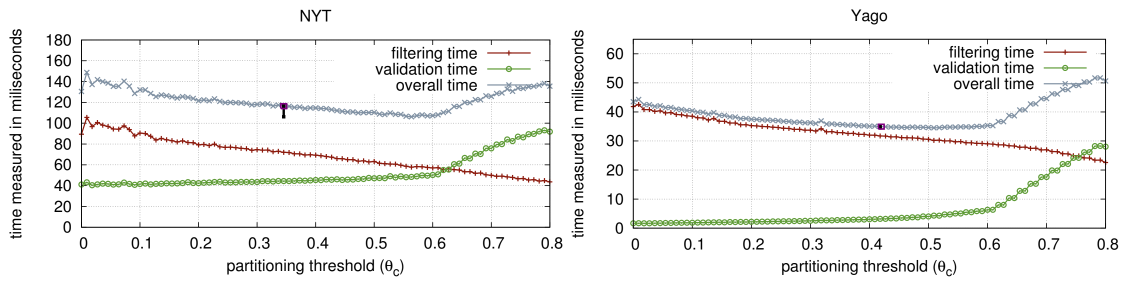


Figure 7: Trend of the filtering and validation time of the coarse index for $k = 10$, $\theta = 0.2$ and varying θ_C. The small rectangle depicts the performance of the coarse index if θ_C was chosen by the model and the vertical line the difference in performance.

ing of blocks. In cases where the best blocks will not result in similar rankings, Blocked+Prune terminates early. Thus, when searching for exact matches, the Blocked+Prune technique performs especially well, outperforming AdaptSearch by a factor of 1.2. Same as for the coarse index, dropping lists further improves the performance of the Blocked+ Prune technique. Increasing the values of θ degrades the performance of all the processing techniques. The Blocked+ Prune+Drop technique performs worse than the F&V+Drop, because sorting the lists adds some overhead to the processing while the pruning is not so effective. The F&V+Drop technique is performing very well, in fact we measured its performance to be very close to the Minimal F&V, especially for small values of θ. Although they are both based on the same concept, F&V+Drop performs better than AdaptSearch, first because it drops one index list more than AdaptSearch, and second, because we are processing relatively short rankings, thus the simple algorithms perform well.

Most of the query processing techniques display the same behavior in the experiments performed on the Yago dataset (Figure 9). What is different here is that none of the processing techniques perform as good as the Minimal F&V, which shows a runtime close to zero. This is due to the fact that the items in the Yago dataset are more equally distributed. In this dataset we have small clusters of similar rankings. However, the clusters seem to be different among them, allowing more rankings to be pruned early on. Moreover, for the Yago dataset the Blocked+Prune technique performs very poorly. We believe this is because the overhead of sorting the index list is too big for the small index size. In fact, we measured that for 35% of the queries sorting of the index lists accounts for a third of the execution time, when $k = 10$. The percentage increases as we increase k. The simple baseline ListMerge technique surprisingly outperforms the coarse index and the AdaptSearch algorithm. We believe that this happens because of the small data size and the size of the rankings. Still, ListMerge does not perform better than the Coarse+Drop technique, except for $\theta = 0.3$ and $k = 10$. For this dataset, the AdaptSearch algorithm shows better performance, performing better than the coarse index in most of the cases. However, the Coarse+Drop technique and some of the hybrid techniques still outperform the competitor, AdaptSearch.

Distance Function Computations

The difference in performance between the Coarse, Coarse+ Drop, F&V+Drop and Blocked+Prune+Drop algorithms can be explained by looking at the number of distance functions calls, shown in Figure 10. We see that for the Yago dataset

	size in MB		**construction time in sec.**	
	NYT	Yago	NYT	Yago
Plain Inverted Index	480	24	3.37	0.03
Augmented Inverted Index	661	38	5.72	0.11
Delta Inverted Index	417	35	3.63	0.086
BK-tree	276	11	1206.75	12.11
M-tree	265	11	35.00	0.47
Coarse Index	367	26	1392.35	19.57

Table 6: Size and construction time of indices for $k = 10$

the final result set is very small, practically almost 1, and the number of distance function computations performed by all the algorithms is significantly larger than the final result set. On the other hand, for the NYT data set—where we have a skewed distribution of the items—the number of false positives is very small, resulting in a very good performance of the F&V+Drop and Blocked+Prune+Drop processing techniques. Combining these with the coarse index even further reduces the number of distance function computations, i.e., the number of distance function computations is smaller than the final result set. This is because for the exact matching rankings in one partition, the Footrule distance is not computed again during query processing time.

7.2 Index Size and Construction Time

In Table 6 the size and the index construction time is shown for both datasets for $k = 10$. Delta Inverted Index is the index used in the AdaptSearch algorithm. For the coarse index, we set $\theta_C = 0.5$. We see that all the indices are smaller than 1GB. All indices store the complete rankings, thus their sizes do not differ significantly. The rank-augmented inverted index requires the most storage as it keeps both the complete rankings, and the position augmented index lists to support different processing techniques.

The construction time of the coarse index is the most expensive one, as we need to build a BK-tree, partition it and add the medoids to the inverted index. The construction of the BK-tree is expensive as the tree is unbalanced and in worst case, we need $O(n^2)$ distance computations. The M-tree index construction time is lower than the BK-tree. Both construction times are worse than the one of the inverted index; creating the inverted index does not imply making any distance computations. However, the construction time of the plain inverted index is cheaper than the augmented one, as we do not consider the position of the rankings.

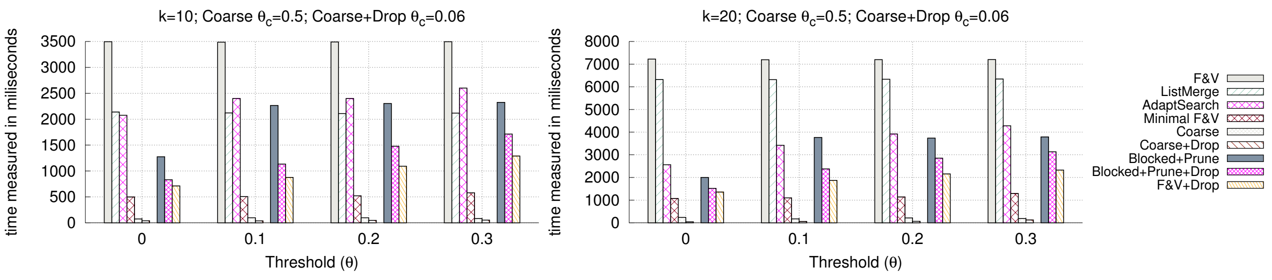


Figure 8: Comparing query processing over coarse index with baseline and competitor approaches (left block) and with other hybrid methods over inverted index (right block) (NYT).

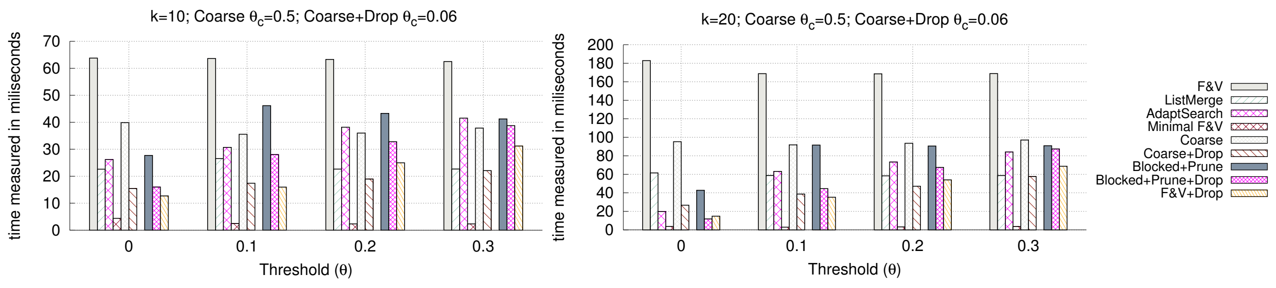


Figure 9: Comparing query processing over coarse index with baseline and competitor approaches (left block) and with other hybrid methods over inverted index (right block) (Yago).

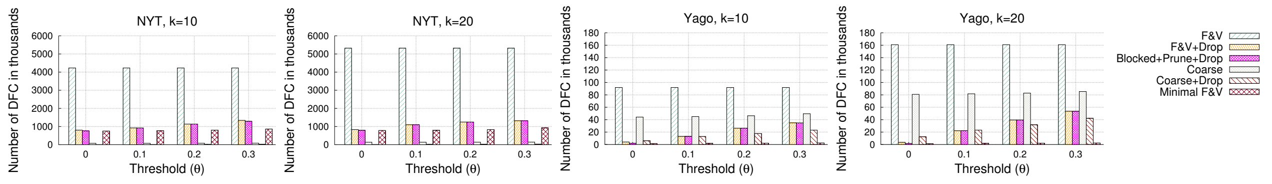


Figure 10: # of distance function calls (DFC) for different query processing methods (Coarse θ_c=0.5; Coarse+Drop $\theta_c = 0.06$)

It is difficult to compare the complexity of the construction time of the different index structures, since the complexity of the metric index structures is usually measured in distance function computation, as this is the most costly operation. On the other hand, in the case of the inverted index there are no distance functions performed during construction at all.

Lessons Learned

Combining the coarse index with the proposed optimizations on the inverted index always leads to performance improvements, independent of the distribution of the items in the dataset. The experiments demonstrate that the Coarse+ Drop technique outperformed state-of-the-art algorithm for similarity search, AdaptSearch, for both datasets. The simple yet accurate model for picking the optimal trade-off point (cf., Section 4) leads close to the best performance of the coarse index. When the query threshold is not known, we can tune the coarse index for the maximum query threshold that we might have. In these cases, the coarse index shows to perform better for a skewed dataset. When having a dataset where the items are unevenly distributed, the F&V+Drop algorithm alone results in huge gains as we only process the smallest index lists. These, as the distribution of the items is skewed, can often contain only few false positives. On the contrary, when the dataset contains chunks of rankings similar to each other, i.e, we have more evenly distributed items, the effect of the early pruning of rankings is most expressed. Thus in these cases, using the Blocked+Prune+Drop algorithm, which combines the early pruning with dropping of entire index lists, leads to the biggest benefits, for small values of θ. Varying the size of the rankings does not have a great impact on the different algorithms. Only when having very small ranking sizes, for instance k=5, the simple baseline ListMerge shows to perform well.

8. CONCLUSION AND OUTLOOK

In this paper, we addressed indexing mechanisms and query processing techniques for ad-hoc similarity search inside sets of rankings. We specifically considered Spearman's Footrule distance for top-k rankings and investigated the trade-offs between metric index structures and inverted indices, known in the literature for indexing set-valued attributes. The presented coarse index synthesizes advantages of metric-space indexing and the ability of inverted indices to immediately dismiss non-overlapping rankings. To understand and automatically tune the necessary partitioning of the rankings, we developed an accurate theoretic cost model; and showed by experiments that it allows reaching performance close to the optimal trade-off point. Further, we presented an algorithm that avoids accessing blocks of an index list during query processing thereby improving performance. We derived upper and lower distance bounds for such an online processing and, further, studied the impact of dropping entire parts of the query depending on the tightness of the query threshold. The presented approaches are to a large extent orthogonal and, by a comprehensive performance evaluation using two real-world datasets, we showed that the individual benefits add up, showing better performance than the competitor, AdaptSearch.

As ongoing work we consider processing large batches of queries, instead of the single ad-hoc queries we addressed in this work. We believe that an approach similar to the coarse indexing can be fruitful here: the query batch can be partitioned into related medoid rankings to prune the search space of potential result rankings.

9. REFERENCES

[1] F. Alvanaki, E. Ilieva, S. Michel, and A. Stupar Interesting event detection through hall of fame rankings. *DBSocial*, 2013.

[2] N. Augsten, A. Miraglia, T. Neumann, and A. Kemper. On-the-fly token similarity joins in relational databases. *SIGMOD*, USA, 2014.

[3] R. A. Baeza-Yates, W. Cunto, U. Manber, and S. Wu. Proximity Matching Using Fixed-Queries Trees. In *CPM*, 1994.

[4] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, and G. Weikum. IO-Top-k: Index-access Optimized Top-k Query Processing. *VLDB*, pages 475–486, 2006.

[5] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling Up All Pairs Similarity Search. *WWW*, pages 131–140, 2007.

[6] R. Bayer and E. M. McCreight. Organization and Maintenance of Large Ordered Indices. *Acta Inf.*, 1, 1972.

[7] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. *SIGMOD*, 1990.

[8] J. L. Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM*, 18(9), 1975.

[9] N. Bruno, S. Chaudhuri, and L. Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *ACM Trans. Database Syst.*, 27(2), 2002.

[10] W. A. Burkhard and R. M. Keller. Some Approaches to Best-match File Searching. *Commun. ACM*, 16(4), 1973.

[11] S. Chaudhuri, V. Ganti, and R. Kaushik. A Primitive Operator for Similarity Joins in Data Cleaning. *ICDE*, 2006.

[12] E. Chávez and G. Navarro. A compact space decomposition for effective metric indexing. *Pattern Recognition Letters*, 26(9), 2005.

[13] E. Chávez, G. Navarro, R. A. Baeza-Yates, and J. L. Marroquín. Searching in metric spaces. *ACM Comput. Surv.*, 33(3), 2001.

[14] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. *VLDB*, 1997.

[15] E. R. D'Avila. M-Tree Implementation at GitHub. `https://github.com/erdavila/M-Tree`.

[16] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *OSDI*, 2004.

[17] C. Dwork, R. Kumar, M. Naor, and D. Sivakumar. Rank aggregation methods for the Web. *WWW*, 2001.

[18] R. Fagin, R. Kumar, and D. Sivakumar. Comparing Top k Lists. *SIAM J. Discrete Math.*, 17(1), 2003.

[19] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4), 2003.

[20] P. Flajolet, D. Gardy, and L. Thimonier. Birthday Paradox, Coupon Collectors, Caching Algorithms and Self-Organizing Search. *Discrete Applied Mathematics*, 39(3), 1992.

[21] V. Ganti, R. Ramakrishnan, J. Gehrke, A. L. Powell, and J. C. French. Clustering Large Datasets in Arbitrary Metric Spaces. *ICDE*, 1999.

[22] S. Helmer and G. Moerkotte. A performance study of four index structures for set-valued attributes of low cardinality. *VLDB J.*, 12(3), 2003.

[23] E. Ilieva, S. Michel, and A. Stupar. The essence of knowledge (bases) through entity rankings. *CIKM*, 2013.

[24] E. H. Jacox and H. Samet. Metric space similarity joins. *ACM Trans. Database Syst.*, 33(2), 2008.

[25] N. Mamoulis. Efficient Processing of Joins on Set-valued Attributes. *SIGMOD*, 2003.

[26] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006.

[27] Y. N. Silva, W. G. Aref, and M. H. Ali. The similarity join database operator. *ICDE*, pages 892–903, 2010.

[28] Y. N. Silva, W. G. Aref, P. Larson, S. Pearson, and M. H. Ali. Similarity queries: their conceptual evaluation, transformations, and processing. *VLDB J.*, 22(3), 2013.

[29] M. Terrovitis, P. Bouros, P. Vassiliadis, T. K. Sellis, and N. Mamoulis. Efficient answering of set containment queries for skewed item distributions. *EDBT*, 2011.

[30] M. Terrovitis, S. Passas, P. Vassiliadis, and T. K. Sellis. A combination of trie-trees and inverted files for the indexing of set-valued attributes. *CIKM*, 2006.

[31] The New York Times Annotated Corpus. `http://corpus.nytimes.com`.

[32] J. K. Uhlmann. Satisfying General Proximity/Similarity Queries with Metric Trees. *Inf. Process. Lett.*, 40(4), 1991.

[33] J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *SIGMOD Conference*, 2012.

[34] Y. Wang, A. Metwally, and S. Parthasarathy. Scalable all-pairs similarity search in metric spaces. *KDD*, 2013.

[35] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient Similarity Joins for Near Duplicate Detection. *WWW*, 2008.

[36] P. N. Yianilos. Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces. *SODA*, 1993.

[37] P. Zezula, P. Savino, G. Amato, and F. Rabitti. Approximate Similarity Retrieval with M-Trees. *VLDB J.*, 7(4), 1998.

[38] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), 2006.

Optimizing Reformulation-based Query Answering in RDF

Damian Bursztyn[§,⋆] François Goasdoué[†,§] Ioana Manolescu[§,⋆]

[§]INRIA, France [⋆]Université Paris-Sud, France [†]Université Rennes 1, France

firstname.lastname [§]@inria.fr [⋆]@lri.fr [†]@irisa.fr

ABSTRACT

Reformulation-based query answering is a query processing technique aiming at answering queries under constraints. It consists of reformulating the query based on the constraints, so that evaluating the reformulated query directly against the data (i.e., without considering any more the constraints) produces the correct answer set.

In this paper, we consider optimizing reformulation-based query answering in the setting of *ontology-based data access*, where SPARQL conjunctive queries are posed against RDF facts on which constraints expressed by an RDF Schema hold. The literature provides query reformulation algorithms for many fragments of RDF. However, reformulated queries may be complex, thus may not be efficiently processed by a query engine; well established query engines even fail processing them in some cases.

Our contribution is (i) to generalize prior query reformulation languages, leading to investigating a *space of reformulated queries* we call `JUCQ`s (joins of unions of conjunctive queries), instead of a single reformulation; and (ii) an effective and efficient *cost-based algorithm* for selecting from this space, the reformulated query with the lowest estimated cost. Our experiments show that our technique enables reformulation-based query answering where the state-of-the-art approaches are simply unfeasible, while it may decrease its cost by orders of magnitude in other cases.

1. INTRODUCTION

The Resource Description Framework (RDF) [1] is a graph-based data model promoted by the W3C as the standard for Semantic Web applications. As such, it comes with an ontology language, *RDF Schema* (RDFS), that can be used to enhance the description of RDF *graphs*, i.e., RDF datasets. The W3C standard for querying RDF graphs is the SPARQL Protocol and RDF Query Language (SPARQL) [2].

Answering SPARQL queries requires to handle the *implicit information* modeled in RDF graphs, through the essential RDF reasoning mechanism called *RDF entailment*. Query answers are defined based on both the explicit and the implicit content of an RDF graph. Thus, ignoring implicit information leads to incomplete answers [2].

Two main methods exist for answering SPARQL queries against RDF graphs, both of which consists of a *reasoning* step, either on the graphs or on queries, followed by a *query evaluation* step. A popular reasoning method is *graph saturation* (a.k.a. *closure*). This consists of pre-computing and adding to an RDF graph all its implicit information, to make it explicit. Answering queries through saturation, then, amounts to evaluating the queries on the saturated graph. While saturation leads to efficient query processing, it requires time to be computed, space to be stored, and must be recomputed upon updates. The alternative reasoning step is *query reformulation*. This consists in turning a query into a *reformulated query*, which, evaluated against a non-saturated RDF graph, yields the exact answers to the original query. Since reformulation takes place at query time, it is intrinsically robust to updates; the query reformulation process in itself is also typically very fast, since it only operates on the query, not on the data. However, reformulated queries are often syntactically more complex than the original ones, thus their evaluation may be costly or even unfeasible.

Saturation-based query answering has been well studied by now; efficient saturation algorithms have been proposed, including incremental ones [3, 4, 5, 6]. Most RDF data management systems use saturation-based query answering, either by providing such a reasoning service on RDF graphs, like 3store, OWLIM, Sesame, etc., or by simply assuming that RDF graphs have been saturated prior to loading. Most systems built on top of relational data management systems (RDBMSs, in short) or RDBMS-style engines [7, 8, 9] fall in this category.

Reformulation-based query answering has also been the topic of many works [6, 10, 11, 12, 13], including ours [4, 14, 15]. Existing techniques apply to the Description Logics (DL) [16] fragment of RDF, the conjunctive subset of SPARQL subset and extensions thereof [10, 11, 14, 15, 17, 18, 19], including the "database fragment" of RDF we introduced in [4], the most expressive RDF fragment to date. Only a few RDF data management systems, such as AllegroGraph, Stardog or Virtuoso, use reformulation, in some cases incomplete. The main reason is that state-of-the-art techniques produce reformulated queries whose evaluation is *inefficient*. A query is typically reformulated into an equivalent *large* union of conjunctive queries (`UCQ`), maximally contained in the original query w.r.t. the RDF Schema constraints [4, 6, 10, 11, 12, 14, 15, 17, 18], or in languages for

10.5441/002/edbt.2015.24

which no well established off-the-shelf query engine exists, such as nested SPARQL [19]. The technique of [13], when translated to the RDF setting, reformulates a conjunctive query into a so-called *semi-conjunctive query* (SCQ), which is a join of unions of atomic queries. While in many cases this performs better than the UCQ reformulations used in prior work, we show that the reformulation of [13] is only another point in a space, in which we automatically identify the most efficient alternative. Finally, a mix of saturation- and reformulation-based query answering has been investigated in [11]. Only RDF Schema contraints are saturated (thus need maintenance), which allows to avoid generating as part of the reformulation, empty-answer subqueries. This may reduce its syntactic size, but (depending on the ability of the underlying engine to detect empty-answers queries early on) the resulting reformulated query may still be hard to evaluate.

This work focuses on *optimizing reformulation-based query answering in RDF.*

We consider the setting in which *conjunctive queries (*CQ*), once reformulated into unions of conjunctive queries (*UCQ*) or semi-conjunctive queries (*SCQ*), are handled for evaluation to a query evaluation engine*, which can be an RDBMS, a dedicated RDF storage and query processing engine, or more generally any system capable of evaluating *selections*, *projections*, *joins* and *unions*. As our experiments show, the evaluation of reformulated queries may be very challenging even for well-established relational or native RDF processors, which may handle them inefficiently or simply fail to handle them, even on moderate-size datasets.

The approach we take is the following. Given a SPARQL conjunctive query q and a query reformulation algorithm $\mathcal{A}$ which turns a CQ into a UCQ, we explore a novel, large *space of alternative reformulations* of q that we term JUCQ (for *joins of unions and conjunctive queries*, and pick the JUCQ reformulation with the lowest estimated cost. Each JUCQ reformulation is obtained based on a carefully chosen set of invocations of the algorithm $\mathcal{A}$, guided by our cost model.

Contributions. The contributions we bring to the problem of efficiently answering SPARQL queries, through reformulation, can be outlined as follows (see Figure 1):

1. We generalize the query reformulation approach, by considering a large *space of alternative (equivalent)* JUCQ *reformulations*. This space corresponds to the yellow-background box in Figure 1; it includes and significantly generalizes the prior works based on UCQ or SCQ reformulation. We characterize the size of our space of alternatives, and show that it is oftentimes too large to be completely explored.
2. We define a *cost model* for estimating the evaluation performance of our reformulated queries through a relational engine; other functions can be used instead, and we show that an RDBMSs' internal cost model can easily be used, too.
3. We devise a novel *algorithm* which selects one alternative reformulated query, namely q^{best} in Figure 1, which (*i*) computes the same result as the UCQ reformulated query q^{ref}, and (*ii*) reduces significantly the query evaluation cost (or simply makes it possible when evaluating the plain reformulation fails!)
4. We implemented this algorithm and deployed it on top of three well-established RDBMSs, which we show differ significantly in their ability to handle UCQ and SCQ reformulations proposed in the previous work. Our experiments confirm that our algorithm *makes the most out of each of these engines* by leveraging their strengths and avoiding their weaknesses thanks to the usage of our cost model, which we calibrate separately for each system. This makes reformulation feasible when UCQ and/or SCQ fail, and brings performance improvements of several orders of magnitude w.r.t. UCQ.
5. Finally, we compare our reformulation-based query answering technique against saturation-based query answering, both through an RDBMS and the native RDF platform Virtuoso. These experiments confirm the robustness and performance of our technique, showing in particular that in some cases its performance approaches that of saturation-based query answering.

In the sequel, Section 2 introduces RDF, SPARQL conjunctive queries, query reformulation and the performance issues raised by the evaluation of reformulated queries. Section 3 characterizes our solution search space and formalizes our problem statement. In Section 4, we present our cost model and solution search space exploration technique, which we evaluate through experiments in Section 5. We discuss related work in Section 6, then we conclude.

2. PRELIMINARIES

In Section 2.1 we introduce *RDF graphs*, modeling RDF datasets. Section 2.2 presents the SPARQL conjunctive queries, a.k.a. *Basic Graph Pattern queries.* In Section 2.3, we recall the query reformulation algorithm from [4] used in the present work, chosen because the RDF fragment it applies to is the largest known to date. However, as previously explained, our optimization technique can use any CQ to UCQ reformulation algorithm among those applicable to RDF.

2.1 RDF Graphs

An *RDF graph* (or *graph*, in short) is a set of *triples* of the form s p o. A triple states that its *subject* s has the *property* p, and the value of that property is the *object* o. We consider only well-formed triples, as per the RDF specification [1], using uniform resource identifiers (URIs), typed or un-typed literals (constants) and blank nodes (unknown URIs or literals).

Blank nodes are essential features of RDF allowing to support *unknown URI/literal tokens.* These are conceptually similar to the variables used in incomplete relational databases based on *V-tables* [20, 21], as shown in [4].

Notations. We use s, p, o and $\_{:}b$ in triples as placeholders. Literals are shown as strings between quotes, e.g., "*string*". Finally, the set of values – URIs (U), blank nodes (B), and literals (L) – of an RDF graph G is denoted Val(G).

Figure 2 (top) shows how to use triples to describe resources, that is, to express class (unary relation) and property (binary relation) assertions. The RDF standard [1] provides a set of built-in classes and properties, as part of the rdf: and rdfs: pre-defined namespaces. We use these namespaces exactly for these classes and properties, e.g., rdf:type specifies the class(es) to which a resource belongs.

Example 1 (**RDF graph**). *The RDF graph* G *below comprises information about a book, identified by* doi_1*: its author (a blank node* $\_{:}b_1$ *related to the author name, which is a literal), title and date of publication.*

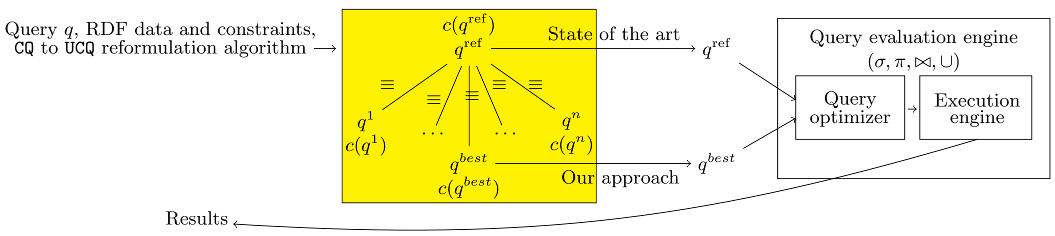


Figure 1: Outline of our approach for efficiently evaluating reformulated SPARQL conjunctive queries.

Assertion	Triple	Relational notation
Class	`s rdf:type o`	$\mathtt{o(s)}$
Property	`s p o`	$\mathtt{p(s, o)}$

Constraint	Triple	OWA interpretation
Subclass	`s rdfs:subClassOf o`	$\mathtt{s} \subseteq \mathtt{o}$
Subproperty	`s rdfs:subPropertyOf o`	$\mathtt{s} \subseteq \mathtt{o}$
Domain typing	`s rdfs:domain o`	$\Pi_{\text{domain}}(\mathtt{s}) \subseteq \mathtt{o}$
Range typing	`s rdfs:range o`	$\Pi_{\text{range}}(\mathtt{s}) \subseteq \mathtt{o}$

Figure 2: RDF (top) & RDFS (bottom) statements.

$$\mathtt{G} = \begin{array}{l}\{\text{doi}_1 \text{ rdf:type Book},\\ \text{doi}_1 \text{ writtenBy } \_{:}b_1,\\ \text{doi}_1 \text{ hasTitle ``Game of Thrones''},\\ \_{:}b_1 \text{ hasName ``George R. R. Martin''},\\ \text{doi}_1 \text{ publishedIn ``1996''}\}\end{array}$$

RDF Schema. A valuable feature of RDF is RDF Schema (RDFS) that allows enhancing the descriptions in RDF graphs. RDFS triples declare *semantic constraints* between the classes and the properties used in those graphs.

Figure 2 (bottom) shows the allowed constraints and how to express them; *domain* and *range* denote respectively the first and second attribute of every property. The RDFS constraints (Figure 2) are interpreted under the open-world assumption (OWA) [20]. For instance, given two relations R_1, R_2, the OWA interpretation of the constraint $R_1 \subseteq R_2$ is: any tuple t in the relation R_1 *is considered as being also in the relation* R_2 (the inclusion constraint *propagates* t *to* R_2). More specifically, when working with the RDF data model, if the triples hasFriend rdfs:domain Person and Anne hasFriend Marie hold in the graph, then so does the triple Anne rdf:type Person. The latter is due to the rdfs:domain constraint in Figure 2.

RDF entailment. *Implicit triples* are an important RDF feature, considered part of the RDF graph even though they are not explicitly present in it, e.g., Anne rdf:type Person above. W3C names *RDF entailment* the mechanism through which, based on a set of explicit triples and some *entailment rules*, implicit RDF triples are derived. We denote by $\vdash^{i}_{\text{RDF}}$ *immediate entailment*, i.e., the process of deriving new triples through a *single* application of an entailment rule. More generally, a triple $\mathtt{s\ p\ o}$ is entailed by a graph $\mathtt{G}$, denoted $\mathtt{G} \vdash_{\text{RDF}} \mathtt{s\ p\ o}$, if and only if there is a sequence of applications of immediate entailment rules that leads from $\mathtt{G}$ to $\mathtt{s\ p\ o}$ (where at each step of the entailment sequence, the triples previously entailed are also taken into account).

Example 2 (**RDFS**). *Assume that the RDF graph* $\mathtt{G}$ *in Example 1 is extended with the following constraints.*

- *books are publications:*
  Book rdfs:subClassOf Publication
- *writing something means being an author:*
  writtenBy rdfs:subPropertyOf hasAuthor

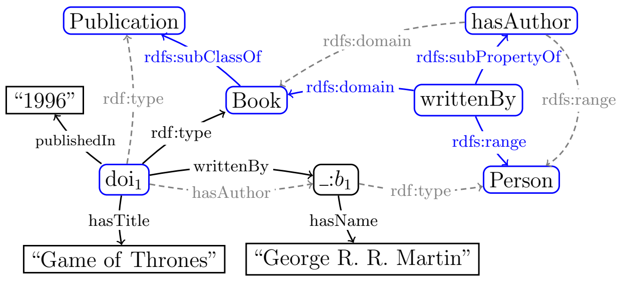


Figure 3: Sample RDF graph.

- *books are written by people:*
  writtenBy rdfs:domain Book
  writtenBy rdfs:range Person

The resulting graph is depicted in Figure 3. Its implicit triples are those represented by dashed-line edges.

Saturation. The immediate entailment rules allow defining the finite *saturation* (a.k.a. closure) of an RDF graph $\mathtt{G}$, which is the RDF graph $\mathtt{G}^\infty$ defined as the fixed-point obtained by repeatedly applying $\vdash^{i}_{\text{RDF}}$ rules on $\mathtt{G}$.

The saturation of an RDF graph is unique (up to blank node renaming), and does not contain implicit triples (they have all been made explicit by saturation). An obvious connection holds between the triples entailed by a graph $\mathtt{G}$ and its saturation: $\mathtt{G} \vdash_{\text{RDF}} \mathtt{s\ p\ o}$ if and only if $\mathtt{s\ p\ o} \in \mathtt{G}^\infty$.

RDF entailment is part of the RDF standard itself; in particular, *the answers to a query posed on* $\mathtt{G}$ *must take into account all triples in* $\mathtt{G}^\infty$, since *the semantics of an RDF graph is its saturation* [2].

2.2 BGP Queries

We consider the well-known subset of SPARQL consisting of (unions of) *basic graph pattern* (BGP) queries, modeling the SPARQL conjunctive queries. Subject of several recent works [4, 22, 23, 24], BGP queries are the most widely used subset of SPARQL queries in real-world applications [24]. A BGP is a set of *triple patterns*, or triples/atoms in short. Each triple has a subject, property and object, some of which can be variables.

Notations. In the following we use the conjunctive query notation $q(\bar{x})$:- $t_1, \ldots, t_\alpha$, where $\{t_1, \ldots, t_\alpha\}$ is a BGP; the query head variables $\bar{x}$ are called *distinguished variables*, and are a subset of the variables occurring in $t_1, \ldots, t_\alpha$; for boolean queries $\bar{x}$ is empty. The head of q is $q(\bar{x})$, and the body of q is $t_1, \ldots, t_\alpha$. We use x, y, z, etc. to denote variables in queries. We denote by $\mathtt{VarBl}(q)$ the set of variables *and* blank nodes occurring in the query q.

Query evaluation. Given a query q and an RDF graph $\mathtt{G}$, the *evaluation of* q *against* $\mathtt{G}$ is:

$$q(\mathtt{G}) = \{\bar{x}_\mu \mid \mu : \mathtt{VarBl}(q) \to \mathtt{Val}(\mathtt{G}) \text{ is a total assignment}$$

$$\text{such that } t_1^\mu \in \mathtt{G}, t_2^\mu \in \mathtt{G}, \ldots, t_\alpha^\mu \in \mathtt{G}\}$$

where we denote by t^μ the result of replacing every occurrence of a variable or blank node $e \in \mathtt{VarBl}(q)$ in the triple t, by the value $\mu(e) \in \mathtt{Val}(\mathtt{G})$.

Note that evaluation *treats the blank nodes in a query exactly as it treats non-distinguished variables* [25]. Thus, in the sequel, without loss of generality, we consider queries where all blank nodes have been replaced by (new) distinct non-distinguished variables.

Query answering. The evaluation of q against G uses only G's explicit triples, thus may lead to an incomplete answer set. The (complete) *answer set* of q against G is obtained by the evaluation of q against $\mathtt{G}^\infty$, denoted by $q(\mathtt{G}^\infty)$.

Example 3 (**Query answering**). *The following query asks for the names of authors of books somehow connected to the literal 1996:*

$q(x_3)$:- x_1 hasAuthor x_2, x_2 hasName x_3, x_1 x_4 "1996"

Its answer against the graph in Figure 3 is $q(\mathtt{G}^\infty) = \{\langle\text{"George R. R. Martin"}\rangle\}$. *The answer results from* $\mathtt{G} \vdash_{\mathrm{RDF}}$ doi$_1$ hasAuthor _:b_1 *and the assignment* $\mu = \{x_1 \leftarrow \text{doi}_1, x_2 \leftarrow \_{:}b_1, x_3 \leftarrow \text{"George R. R. Martin"}, x_4 \leftarrow \text{publishedIn}\}$. *Observe that evaluating q directly against* G *leads to the empty answer, which is obviously incomplete.*

2.3 Query answering against RDF databases

The *database (DB)* fragment of RDF [4] is, to the best of our knowledge, the most expressive RDF fragment for which both *saturation-* and *reformulation-based RDF query answering* has been defined and practically experimented. The fragment is thus named due to the fact that query answering against any graph from this fragment, called an *RDF database*, can be easily implemented on top of any RDBMS. This DB fragment is defined by:

- *Restricting RDF entailment* to the RDF Schema constraints only (Figure 2), a.k.a. RDFS entailment. Consequently, the DB fragment focuses only on the application domain knowledge, a.k.a. ontological knowledge, and not on the RDF meta-model knowledge which mainly begets high-level typing of subject, property and object values found in triples with abstract RDF built-in classes, e.g., rdf:Resource, rdfs:Class, etc.
- *Not restricting RDF graphs in any way.* In other words, any triple allowed by the RDF specification is also allowed in the DB fragment.

Saturation-based query answering amounts to precomputing the saturation of a database db using its RDFS constraints in a forward-chaining fashion, so that the *evaluation* of every incoming query q against the saturation yields the correct answer set [4]: $q(\mathtt{db}^\infty) = q(\textbf{Saturate}(\mathtt{db}))$. This technique follows directly from the definitions in Section 2.1 and Section 2.2, and the W3C's RDF and SPARQL recommendations.

Reformulation-based query answering, in contrast, leaves the database db untouched and reformulates every incoming query q using the RDFS constraints in a backward-chaining fashion, $\textbf{Reformulate}(q, \mathtt{db}) = q^{\mathrm{ref}}$, so that the *relational evaluation* of this reformulation against the (non-saturated) database yields the correct answer set [4]: $q(\mathtt{db}^\infty) = q^{\mathrm{ref}}(\mathtt{db})$. The **Reformulate** algorithm, introduced in [23] and extended in [4], exhaustively applies a set of 13 reformulation rules. Starting from the incoming BGP query q to answer against db, the algorithm produces a *union of BGP queries* retrieving the correct answer set from the database, even if the latter is not saturated.

Triple	#answers	#reformulations	#answers after reformulation
(t_1)	18,999,082	188	33,328,108
(t_2)	0	4	3,223
(t_3)	396	3	683

Table 1: Characteristics of the sample query q_1.

Example 4 (**Query reformulation**). *The reformulation of* $q(x,y)$:- x rdf:type y *w.r.t. the database* db *(obtained from the RDF graph* G *depicted in Figure 3), asking for all resources and the classes to which they belong, is:*

(0)	$q(x,y)$:- x rdf:type y	$\cup$
(1)	$q(x,\text{Book})$:- x rdf:type Book	$\cup$
(2)	$q(x,\text{Book})$:- x writtenBy z	$\cup$
(3)	$q(x,\text{Book})$:- x hasAuthor z	$\cup$
(4)	$q(x,\text{Publication})$:- x rdf:type Publication	$\cup$
(5)	$q(x,\text{Publication})$:- x rdf:type Book	$\cup$
(6)	$q(x,\text{Publication})$:- x writtenBy z	$\cup$
(7)	$q(x,\text{Publication})$:- x hasAuthor z	$\cup$
(8)	$q(x,\text{Person})$:- x rdf:type Person	$\cup$
(9)	$q(x,\text{Person})$:- z writtenBy x	$\cup$
(10)	$q(x,\text{Person})$:- z hasAuthor x	

The terms (1), (4) *and* (8) *result from* (0) *by instantiating the variable y with classes from* db*, namely* {Book, Publication, Person}*. Item* (5) *results from* (4) *by using the subclass constraint between books and publications.* (2), (6) *and* (9) *result from their direct predecessors in the list, and are due to the domain and range constraints. Finally,* (3), (7) *and* (10) *result from their direct predecessors and the sub-property constraint present in the database.*

Evaluating this reformulation against db *returns the same answer as* $q(\mathtt{G}^\infty)$*, i.e., the answer set of q.*

3. OPTIMIZED REFORMULATION

We first introduce, by examples, the performance issues raised by the evaluation of state-of-the-art reformulated queries. We then introduce our novel reformulation search space and formalize our optimization problem.

Motivating Example 1. *Consider the three triples query q_1 shown below:*

$$\begin{array}{lll} q_1(x,y) \text{ :- } & x \text{ rdf:type } y, & (t_1) \\ & x \ \mathit{ub{:}degreeFrom} \ \text{“}\mathit{http://www.Univ532.edu}\text{”}, & (t_2) \\ & x \ \mathit{ub{:}memberOf} \ \text{“}\mathit{http://www.Dept1.Univ7.edu}\text{”} & (t_3) \end{array}$$

Table 1 gives some intuition on the difficulty of answering q_1 over an 10^8 triples LUBM [26] benchmark dataset:

The state-of-the-art CQ *to* UCQ *reformulation-based query answering needs to evaluate a reformulated query q_1', which is a union of* 2,256 *conjunctive queries, each of which consists of three triples (one for the reformulation of each triple in the original q_1). This query appears in Table 2, where all the triples t_1, t_2, t_3 are reformulated together by a* CQ *to* UCQ *reformulation algorithm denoted $(.)^{ref}$. Observe that in q_1', many sub-expressions are repeated; for instance, the join over the single triples resulting from the reformulation of triples (t_2) and (t_3) will appear for each of the* 188 *reformulations of triple (t_1). Evaluating q_1' on the* 100 *million triples LUBM dataset takes more than* ***6 seconds****, in the same experimental setting.*

	Joins of UCQs	#reformulations	exec.time (ms)
q_1'	$(t_1,t_2,t_3)^{ref}$	2,256	6,387
q_1''	$(t_1)^{ref} \bowtie (t_2)^{ref} \bowtie (t_3)^{ref}$	195	1,074,026
	$(t_1,t_2)^{ref} \bowtie (t_3)^{ref}$	755	1,968
	$(t_1)^{ref} \bowtie (t_2,t_3)^{ref}$	200	846,710
q_1'''	$(t_1,t_3)^{ref} \bowtie (t_2)^{ref}$	568	554
	$(t_1,t_2)^{ref} \bowtie (t_1,t_3)^{ref}$	1,316	2,734
	$(t_1,t_2)^{ref} \bowtie (t_2,t_3)^{ref}$	764	2,289
	$(t_1,t_3)^{ref} \bowtie (t_2,t_3)^{ref}$	576	588

Table 2: Sample reformulations of q_1.

Triple	#answers	#reformulations	#answers after reformulation
(t_1)	18,999,082	188	33,328,108
(t_2)	18,999,082	188	33,328,108
(t_3)	476	1	476
(t_4)	509	1	509
(t_5)	7,299,701	3	7,803,096
(t_6)	7,299,701	3	7,803,096

Table 3: Characteristics of the sample query q_2.

Alternatively, one could consider the equivalent query $q_1'' = (t_1)^{ref} \bowtie (t_2)^{ref} \bowtie (t_3)^{ref}$*, which joins the* CQ *to* UCQ *reformulation of each query's triple. In other terms,* q_1'' *first reformulates each triple (into, respectively, a union of* 188*,* 4*, and* 3 *queries), and then joins these unions. This query corresponds to the simple semi-conjunctive queries (*SCQ*) alternative proposed in [13]. While this avoids the repeated work, its performance is much worse: it takes about* ***1074 seconds*** *to evaluate.*

Let us now consider the following equivalent query $q_1''' = (t_1,t_3)^{ref} \bowtie (t_2)^{ref}$ *where* t_1, t_2, t_3 *are the triples of the query* q_1*. Evaluating* q_1''' *in the same experimental setting takes* ***554 ms****, more than* ***10 times faster*** *than the initial reformulation. The performance improvement of* q_1''' *over* q_1'' *is due to the intelligent grouping of the triples* t_1 *and* t_3 *together. Such grouping of triples reduce the cardinality of the respective reformulated queries. Thus,* $(t_1,t_3)^{ref}$ *has* 2,045 *answers and* 564 *reformulations. Table 2 shows the number of reformulations and execution time for all the eight possible combinations of triples.*

Motivating Example 2. *Consider now the six triples query* q_2 *shown below:*

$q_2(x,u,y,v,z)$:-
- x rdf:type u, (t_1)
- y rdf:type v, (t_2)
- x *ub:mastersDegreeFrom* "$http://www.Univ532.edu$", (t_3)
- y *ub:doctoralDegreeFrom* "$http://www.Univ532.edu$", (t_4)
- x *ub:memberOf* z (t_5)
- y *ub:memberOf* z (t_6)

Statistics on the query triples, when evaluated over a 100 *million triples LUBM dataset, appear in Table 3. The* CQ *to* UCQ *reformulation of* q_2*, on the other hand, leads to a query* q_2' *corresponding to a union of* 318,096 *six triples queries. Due to its complexity,* q_2' ***could not be evaluated*** *in the same experimental setting*[1].

[1]Concretely, a *stack depth limit exceeded error* was thrown by the DBMS. Further, other queries presented I/O exceptions thrown by the DBMS, in connection with a failed attempt to materialize an intermediary result. While it may be possible to tune some parameters to make the evaluation of such queries possible, the same error was raised by many large-reformulation queries, a signal that their peculiar shape is problematic.

Now consider the query $q_2'' = (t_1)^{ref} \bowtie (t_2)^{ref} \bowtie (t_3)^{ref} \bowtie (t_4)^{ref} \bowtie (t_5)^{ref} \bowtie (t_6)^{ref}$*, where* $t_1, \ldots, t_6$ *are the triples of* q_2*; again, this corresponds to the* SCQ *reformulation proposed in [13].* q_2'' *is equivalent to* q_2^{ref}*, and in the same experimental setting, it is evaluated in* ***229 seconds****. This is due to the large results of the (syntactically smal) subqueries* $(t_1)^{ref}, \ldots, (t_6)^{ref}$ *(especially the first two, each with* 33,328,108 *results), which required some time to join.*

Finally, consider the query $q_2''' = (t_1,t_3)^{ref} \bowtie (t_3,t_5)^{ref} \bowtie (t_2,t_4)^{ref} \bowtie (t_4,t_6)^{ref}$*, also equivalent to* q_2'*. Evaluating* q_2''' *takes* ***524 ms****, more than* ***430 times faster*** *than one-triple reformulation. As in the previous example,* q_2''' *gains over* q_2'' *by first, reducing repeated work, and second, intelligently grouping triples so that the query corresponding to each triple group can be efficiently evaluated and returns a result of manageable size. In particular, the biggest-size triples* (t_1) *and* (t_2) *had been grouped with* (t_3) *and* (t_4) *respectively, resulting in smaller intermediate results of* 2,296 *and* 2,475 *rows respectively, and improving the perfomance. Grouping triples* (t_3) *and* (t_4) *with the* (t_5) *and* (t_6) *respectively, yields analogous performance improvements.*

As the above examples illustrate, generalizing the state-of-the-art query reformulation language of UCQs [4, 6, 10, 11, 12, 14, 15, 17, 18] or of SCQs [13], to that of *joins* of UCQs, offers a great potential for improving the performance of reformulated queries. We introduce:

Definition 3.1 (JUCQ). *A* Join of Unions of Conjunctive Queries (JUCQ) *is defined as follows:*
- *any conjunctive query (*CQ*) is a* JUCQ*;*
- *any union of* CQ*s (*UCQ*) is a* JUCQ*;*
- *any join of* UCQ*s is* JUCQ.

In this work, we address the challenge of finding the best-performing JUCQ *reformulation* of a BGP query against an RDF database, among those that can be derived from a *query cover.* We define these notions as follows:

Definition 3.2 (JUCQ reformulation). *A* JUCQ reformulation $q^{\mathtt{JUCQ}}$ *of a BGP query* q *w.r.t. a database* $\mathtt{db}_1$ *is a* JUCQ *such that* $q^{\mathtt{JUCQ}}(\mathtt{db}_2) = q(\mathtt{db}_2^\infty)$*, for any RDF database* $\mathtt{db}_2$ *having the same schema as* $\mathtt{db}_1$.

Recall that two RDF databases have the same schema iff their saturations have the same RDFS statements.

BGP query covering is a technique we introduce for exploring a space of JUCQ reformulations of a given query. The idea is to *cover* a query q with (possibly overlapping) subqueries, so as to produce a JUCQ reformulation of q by joining the (state-of-the-art) CQ to UCQ reformulations of these subqueries, obtained through any reformulation algorithm in the literature (e.g., [4]). Formally:

Definition 3.3 (BGP query cover). *A* cover *of a BGP query* $q(\bar{x})$:- $t_1, \ldots, t_n$ *is a set* $C = \{f_1, \ldots, f_m\}$ *of non-empty subsets of* q*'s triples, called* fragments*, such that* $\bigcup_{i=1}^m f_i = \{t_1, \ldots, t_n\}$*, no fragment is included into another, i.e.,* $f_i \not\subseteq f_j$ *for* $1 \le i,j \le m$ *and* $i \neq j$*, and: if* C *consists of more than 1 fragment, then any fragment joins at least with another, i.e., they share a variable.*

For example, a cover of our query q_1 is $\{\{t_1,t_2\},\{t_2,t_3\}\}$.

Definition 3.4 (Cover queries of a BGP query). *Let* $q(\bar{x})$:- $t_1, \ldots, t_n$ *be a BGP query and* $C = \{f_1, \ldots, f_m\}$ *one of its covers. A* cover query $q_{|f_i, 1 \le i \le m}$ *of* q *w.r.t.* C

is the subquery whose body consists of the triples in f_i and whose head variables are the distinguished variables $\bar{x}$ of q appearing in the triples of f_i, plus *the variables appearing in a triple of f_i that are shared with some triple of another fragment $f_{j,1\leq j\leq m,j\neq i}$, i.e., on which the two fragments join.*

For example, the cover $\{\{t_1\},\{t_2,t_3\}\}$ of our query q_1 leads to the cover queries $q_{|f_1}(x,y)$:- x rdf:type y, and $q_{|f_2}(x)$:- x *ub:degreeFrom* "*http* : //*www.Univ*532.*edu*", x *ub:memberOf* "*http* : //*www.Dept*1.*Univ*7.*edu*".

Query evaluation through an RDBMS is typically much more efficient when all the atoms of the query are connected through joins (in which case, properly optimized queries oftentimes run in linear time in the size of the data), than when the query comprises a cartesian product (which leads to unavoidable quadratic or higher complexity in the size of the data). Therefore, in this work, we only consider fragments which do not feature a cartesian product.

The theorem below states that evaluating a query q as the join of the cover queries resulting from one of its covers, yields the answer set of q:

Theorem 3.1 (Cover-based reformulation). *Let $q(\bar{x})$:- $t_1,\ldots,t_n$ be a BGP query and $C=\{f_1,\ldots,f_m\}$ be any of its covers,*

$$q^{\texttt{JUCQ}}(\bar{x})\text{:- } q^{\texttt{UCQ}}_{|f_1}\bowtie\cdots\bowtie q^{\texttt{UCQ}}_{|f_m}$$

is a JUCQ *reformulation of q w.r.t. any database* db*, where every $q^{\texttt{UCQ}}_{|f_i}$ is a* UCQ *reformulation of the cover query $q_{|f_i}$, for $1\leq i\leq m$.*

An *upper bound* on the size of the cover-based reformulation space for a given query of n triples is given by the number of *minimal covers* of a set $\mathcal{S}$ of n elements [27], i.e., a set of non-empty subsets of $\mathcal{S}$ whose union is $\mathcal{S}$, and whose union of all these subsets but one is not $\mathcal{S}$. This bound grows rapidly as the number n of triples in a query's body increases, e.g., 1 for $n=1$, 49 for $n=4$, 462 for $n=5$, 6424 for $n=6$ (`http://oeis.org/A046165`). In practice, however, we require each fragment to share a variable with another (if any), so that cover queries, hence *cover-based reformulations do not feature cartesian products.* Therefore, the number of cover-based reformulations is smaller than the number of minimal covers.

In order to select the best performing cover-based reformulation within the above space, we assume given a *cost function* c which, for a JUCQ q, returns the cost $c(q(\texttt{db}))$ of evaluating it through an RDBMS storing the database db. Function c may reflect any (combination of) query evaluation costs, such as I/O, CPU etc. As customary, we rely on a *cost estimation* function c^{ϵ}, which statically provides an approximate value of c. For simplicity, in the sequel we will use c to denote the estimated cost.

The problem we study can now be stated as follows:

Definition 3.5 (Optimization problem). *Let* db *be an RDF database and q be a BGP query against it. The* optimization problem *we consider is to find a* JUCQ *reformulation $q^{\texttt{JUCQ}}$ of q w.r.t.* db*, among the cover-based reformulations of q with lowest (estimated) cost.*

Optimized queries vs. optimized plans. As stated above and illustrated in Figure 1, we seek the best *query* that is an optimized reformulation of q against db; we do not seek to optimize its *plan*, instead, we take advantage of existing query evaluation engines for optimizing and executing it. Alternatively, one could have placed this study *within an evaluation engine* and investigate *optimized plans.* We comment more the two alternatives in Section 6.

4. EFFICIENT QUERY ANSWERING

We present now the ingredients for setting up our cost-based query answering technique. We introduce, in Section 4.1, our cost model for JUCQ reformulation evaluation through an RDBMS. We then provide, in Section 4.2, an exhaustive algorithm that traverses the search space of reformulated queries, looking for a cover-based reformulation with lower cost. Finally, in Section 4.3, we introduce a greedy, anytime algorithm that outputs a best query cover of the input BGP query, found so far. This one is then used to evaluate the query as stated by Theorem 3.1.

4.1 Cost model

In this section we detail the cost of evaluating a JUCQ (reformulation) sent to an RDBMS. Such a query is a join of UCQs subqueries of the form: $q^{\texttt{JUCQ}}(\bar{x})$:- $q^{\texttt{UCQ}}_1\bowtie\cdots\bowtie q^{\texttt{UCQ}}_m$.

The evaluation cost of $q^{\texttt{JUCQ}}$ is

$$c(q^{\texttt{JUCQ}})=c_{\texttt{db}}+\sum_{q_i^{\texttt{UCQ}}\in q^{\texttt{JUCQ}}}(c_{eval}(q_i^{\texttt{UCQ}})+c_{join}(q^{\texttt{UCQ}}_{i,1\leq i\leq m})+ c_{mat}(q^{\texttt{UCQ}}_{i,1\leq i\leq m,i\neq k}))+c_{unique}(q^{\texttt{JUCQ}})\quad(1)$$

reflecting:

(*i*) the fixed overhead of connecting to the RDBMS $c_{\texttt{db}}$;
(*ii*) the cost to *evaluate* each of its UCQ sub-queries $q_i^{\texttt{UCQ}}$;
(*iii*) the cost of eliminating duplicate rows from each of its UCQ sub-query results;
(*iv*) the cost to *join* these sub-query results;
(*v*) the *materialization* costs: the SQL query corresponding to a JUCQ may have many sub-queries. At execution time, some of these subqueries will have their results materialized (i.e., stored in memory or on disk) while at most one sub-query will be executed in pipeline mode. We assume without loss of generality, that the largest-result sub-query, denoted $q_k^{\texttt{UCQ}}$, is the one pipelined (this assumption has been validated by our experiments so far); and
(*vi*) the cost of eliminating duplicate rows from the result.

In the above, duplicates are eliminated because existing reformulation algorithms (and accordingly, our work) operate under *set semantics*.

Notations. For a given query q over a database db, we denote by $|q|_t$ the estimated number of tuples in q's answer set. Recall that $q_{|\{t_i\}}$ stands for the restriction of q to its i-th triple. Using the notations above, the number of tuples in the answer set of $q_{|\{t_i\}}$ is denoted $|q_{|\{t_i\}}|_t$.

Duplicate elimination costs are estimated using well-known textbook formulas [28]; more details appear in [29].

UCQ evaluation costs are estimated by summing up the estimated costs of the CQs:

$$c_{eval}(q_i^{\texttt{UCQ}})=c_{unique}(q_i^{\texttt{UCQ}})+\sum_{q^{\texttt{CQ}}\in q_i^{\texttt{UCQ}}}c_{eval}(q^{\texttt{CQ}})$$

The cost of evaluating *one* conjunctive query $c_{eval}(q^{\texttt{CQ}})$, where $q^{\texttt{CQ}}(\bar{x})$:- $t_1,\ldots,t_n$, through the RDBMS is made of the *scan* cost for retrieving the tuples for each of its triples, and the cost of *joining* these tuples:

$$c_{eval}(q^{\texttt{CQ}})=c_{scan}(q^{\texttt{CQ}})+c_{join}(q^{\texttt{CQ}})$$

We estimate the *scan cost* of $q^{\texttt{CQ}}$ to:

$$c_{scan}(q^{\texttt{CQ}}) = c_t \times \sum_{t_i \in q^{\texttt{CQ}}} |q^{\texttt{CQ}}_{|\{t_i\}}|_t$$

where c_t is the fixed cost of retrieving one tuple.

The *join* cost of $q^{\texttt{CQ}}$ represents the respective CPU and I/O effort; assuming efficient join algorithms such as hash- or merge-based etc. are available [28], this cost is linear in the total size of its inputs:

$$c_{join}(q^{\texttt{CQ}}) = c_j \times \sum_{t_i \in q^{\texttt{CQ}}} |q^{\texttt{CQ}}_{|\{t_i\}}|_t$$

Therefore, we have:

$$c_{eval}(q_i^{\texttt{UCQ}}) = (c_t + c_j) \times \sum_{q^{\texttt{CQ}} \in q_i^{\texttt{UCQ}}} \sum_{t_i \in q^{\texttt{CQ}}} |q^{\texttt{CQ}}_{|\{t_i\}}|_t \quad (2)$$

UCQ join cost. As before, we consider the join cost to be linear in the total size of its inputs:

$$c_{join}(q^{\texttt{UCQ}}_{i,1\leq i \leq m}) = c_j \times \sum_{q_i^{\texttt{UCQ}}} \sum_{q^{\texttt{CQ}} \in q_i^{\texttt{UCQ}}} \sum_{t_i \in q^{\texttt{CQ}}} |q^{\texttt{CQ}}_{|\{t_i\}}|_t \quad (3)$$

UCQ materialization cost. Finally, we consider the materialization cost associated to a query q is $c_m \times |q|_t$ for some constant c_m:

$$c_{mat}(q^{\texttt{UCQ}}_{i,1\leq i \leq m, i\neq k}) = c_m \times \sum_{q_i^{\texttt{UCQ}}, i\neq k} \sum_{q^{\texttt{CQ}} \in q_i^{\texttt{UCQ}}} \sum_{t_i \in q^{\texttt{CQ}}} |q^{\texttt{CQ}}_{|\{t_i\}}|_t \quad (4)$$

where $q_k^{\texttt{UCQ}}$ is the largest-result sub-query, and the one which is picked for pipelining (and thus not materialized).

Injecting the equations 2, 3 and 4 into the global cost formula 1 leads to the estimated cost of a given JUCQ. This formula relies on estimated cardinalities of various subqueries of the JUCQ, as well as on the system-dependent constants $c_{\texttt{db}}$, c_{scan}, c_{join} and c_{mat}, which we determine by running a set of simple calibration queries on the RDBMS being used. The details are straightforward and we omit them here.

4.2 Exhaustive query cover algorithm (ECov)

As a yardstick for the *quality* of the query covers we find, we developed an exhaustive query cover finding algorithm, called ECov, that traverses the search space of reformulated queries and outputs a query cover leading to a cover-based reformulation with lowest cost.

Given a BGP query q and a database db, ECov enumerates all the possible query covers, estimates the cost of the corresponding cover-based reformulations, and returns a query cover with the lowest estimated cost. We use this cover as "golden standard", i.e., the best solutions based on our cost estimation function,

4.3 Greedy query cover algorithm (GCov)

We now present our optimized query cover finding algorithm (GCov). Intuitively, GCov attempts to identify query covers such that the estimated evaluation cost of each cover fragment (once reformulated), together with the estimated cost of joining the results of these reformulated fragments, is minimized. Performance benefits in this context are attained from two sources: (*i*) avoiding the explosion in the size of the reformulated queries that results when many triples, each having many reformulations, are in the same fragment, and (*ii*) avoiding reformulated fragments with very large results, since materialising and joining them is costly. The key intuition for reaching these goals is to *include highly selective, few-reformulations triples in several cover fragments.* Observe that this is different from (and orthogonal to) join ordering, which the underlying query evaluation engine (RDBMS in this study) applies independently to each reformulated subquery.

Algorithm 1: Greedy query cover algorithm (**GCov**)

```
Input  : BGP query $q(\bar{x}\text{:- } t_1, \ldots, t_n)$, database db
Output: Cover $C_{best}$ for the BGP query $q$
1  $C_0 \leftarrow \{\{t_1\}, \{t_2\}, \ldots, \{t_n\}\}$;
2  $T \leftarrow \{t_1, t_2, \ldots, t_n\}$;
3  $C_{best} \leftarrow C_0$; $moves \leftarrow \emptyset$; $analysed \leftarrow \emptyset$;
4  foreach $f \in C_0, t \in T$ s.t. $t \notin f \wedge connected(f, \{t\})$
     $\wedge\, C_0.add(f,t) \notin analysed$ do
5      $analysed \leftarrow analysed \cup C_0.add(f,t)$;
6      if $C_0.add(f,t)$ est. cost $\leq C_{best}$ est. cost then
7          $moves \leftarrow moves \cup (C_0, f, t)$;
8  while $moves \neq \emptyset$ do
9      $(C, f, t) \leftarrow moves.head()$;
10     $C' \leftarrow C.add(f,t)$;
11     if $C'$ est. cost $\leq C_{best}$ est. cost then
12         $C_{best} \leftarrow C'$;
13         foreach $f \in C', t \in T$ s.t. $t \notin f \wedge$
           $connected(f, \{t\}) \wedge C'.add(f,t) \notin analysed$ do
14             $analysed \leftarrow analysed \cup C'.add(f,t)$;
15             if $C'.add(f,t)$ estimated cost $< C_{best}$
               estimated cost then
16                 $moves \leftarrow moves \cup (C', f, t)$;
17 return $C_{best}$;
```

GCov (Algorithm 1) starts with a simple cover C_0 consisting of *one triple fragments*, and explores possible *moves* starting from this state. A *move* consists of adding to one fragment, an extra triple connected to it by at least one join variable, such that the estimated cost associated to the cover-based reformulation thus obtained is smaller than that before the addition. A move may reduce the cost in two ways: (*i*) by making a fragment more selective, and/or (*ii*) by leading to the removal of some fragments from the cover. For instance, let $\{\{t_1,t_2\},\{t_1,t_3\},\{t_3,t_4\}\}$ be a cover of a four-triples query. The move which adds t_4 to the first fragment, also renders $\{t_3,t_4\}$ redundant. Thus, the cover resulting from the move is: $\{\{t_1,t_2,t_4\},\{t_1,t_3\}\}$.

Possible moves based on the initial cover C_0 are developed and added to the list *moves*, sorted in the increasing order of the estimated cost their bring. Next (line 8), GCov starts exploring possible moves. It picks the most promising one from the sorted *moves* list and applies it, leading to a new query cover C'. If its estimated cost is smaller than the best (least) cost encountered so far, the best solution is updated to reflect this C' (line 12), and possible moves based on C' are developed and added to the sorted *moves* list.

GCov explores query covers in breadth-first and *greedy* fashion, adding to the *moves* list the possible moves starting from the current best cover, and selecting the next move with smallest cost. In practice, one could easily change the stop condition, for instance to return the best found cover as soon as its cost has diminished by a certain ratio, or after a time-out period has elapsed etc.

5. EXPERIMENTAL EVALUATION

We now present an experimental assessment of our ap-

LUBM q	Q_{01}	Q_{02}	Q_{03}	Q_{04}	Q_{05}	Q_{06}	Q_{07}	Q_{08}	Q_{09}	Q_{10}	Q_{11}	Q_{12}	Q_{13}	Q_{14}	Q_{15}
$\lvert q^{ref}\rvert$	136	136	34	564	2	188	156	12	8,496	13	1	1	2	376	3,384
$\lvert q(\mathtt{db})\rvert$ (1M)	123	123	41	26,048	982	5,537	0	269	0	47,268	1,530	88	4,041	20,205	0
$\lvert q(\mathtt{db})\rvert$ (100M)	123	123	41	2,432,964	92,026	523,319	0	269	0	4,409,039	142,337	7,773	376,792	1,883,960	0

LUBM q	Q_{16}	Q_{17}	Q_{18}	Q_{19}	Q_{20}	Q_{21}	Q_{22}	Q_{23}	Q_{24}	Q_{25}	Q_{26}	Q_{27}	Q_{28}
$\lvert q^{ref}\rvert$	2	1	940	2,444	4	1	1	752	52	156	2,256	156	318,096
$\lvert q(\mathtt{db})\rvert$ (1M)	5,364	5,388	47,348	60,342	228,086	60,342	16,134	100	12	19	5	1	0
$\lvert q(\mathtt{db})\rvert$ (100M)	501,063	503,395	4,425,553	5,632,454	2,128,9440	5,632,454	1,510,695	11,820	1,508	1,463	5	1	495

DBLP q	Q_{01}	Q_{02}	Q_{03}	Q_{04}	Q_{05}	Q_{06}	Q_{07}	Q_{08}	Q_{09}	Q_{10}
$\lvert q^{ref}\rvert$	684	292	1,387	1,387	4	19	19	1,721	361	1,923,349
$\lvert q(\mathtt{db})\rvert$	4,898	16,424	5,259,462	60,900	19,576	9,562	9,562	203,462	20	80

Table 4: Characteristics of the queries used in our study.

proach. Section 5.1 describes the experimental settings. Section 5.2 studies the effectiveness and efficiency of our optimized reformulation-based query answering technique. Section 5.3 widens our comparison to saturation-based query answering, then we conclude. For space reasons, more experiment descriptions are relegated to [29].

5.1 Settings

Software. We implemented our reformulation-based query answering framework in Java 7, on top of three well-known RDBMSs, namely: PostgreSQL v9.3.2, System A (last available free edition version), and System B (last available free edition version). For each RDBMS, we instantiated the cost formulas introduced in Section 4.1 with the proper coefficients, learned by running our calibration queries on that system.

Hardware. All the RDMBSs run on 8-core Intel Xeon (E5506) 2.13 GHz machines with 16GB RAM, using Mandriva Linux release 2010.0 (Official).

Data. We conducted experiments using DBLP (8 million triples) [30] and LUBM [26] with 1 and 100 millions triples.

In our experiments, RDFS constraints are kept in memory, while RDF facts are stored in a `Triples(s, p, o)` table, indexed by all permutations of the `s, p, o` columns, leading a total of 6 indexes. Our indexing choice is inspired by [8, 9], to give the RDBMS efficient query evaluation opportunities. Further, as in [4, 8, 9, 23], for efficiency, the `Triples(s, p, o)` table's data are dictionary-encoded, using a unique integer for each distinct value (URIs and literals). The dictionary is stored as a separate table, indexed both by the code and by the encoded value.

Queries. We used 28 and 10 BGP queries for our evaluation on LUBM and DBLP data sets, respectively. The queries can be found in [29], while their main characteristics (number of union terms in their `UCQ` reformulation, denoted $|q^{ref}|$, as well as the number of results when evaluated on our data sets) are shown in Table 4.

Some queries are modified versions of LUBM benchmark queries, in order to remove redundant triples[2]. We designed the others so that (*i*) they have an intuitive meaning, (*ii*) they exhibit a variety of result cardinalities, (*iii*) they exhibit a variety of reformulations, some of which are syntactically complex, to allow a study of the performance issues involved and (*iv*) none of their triples is redundant.

[2]A query triple is redundant when it can be inferred from the others based on the RDFS constraints. For instance, when looking for x such that x is a person and x has a social security number, if we know that only people have such numbers, the triple "x is a person" is redundant.

All measured times are averaged over 3 (warm) executions. Moreover, queries whose evaluation requires *more than 2 hours* were interrupted; we point them out when commenting on the experiments' results.

5.2 Optimized reformulation

In this section, we compare our reformulation-based query answering technique with those from the literature based on `UCQ`s and `SCQ`s.

Effectiveness: is an optimizer needed? The first question we ask is whether exploring the space of `JUCQ` alternatives is actually needed, or could one just rely on a simple (fixed) query cover?

The `UCQ` reformulation used in many prior works is a particular case of the `JUCQ` reformulations we introduced in this work; it corresponds to a cover of a single fragment made of all the query triples (recall q_1' in **Motivating Example 1**, Section 3). From a database perspective, it corresponds to *pushing the joins below a single (potentially large) union.* At the other extreme, the `SCQ` reformulation proposed in [13] is a particular case of `JUCQ` reformulation obtained from a cover where each query triple is alone in a fragment (recall q_1'' in the same example). The `SCQ` reformulation can thus be thought of as *pushing all unions below a the joins.* Both the `UCQ` and `SCQ` reformulations correspond to a cover where *each triple appears in exactly one fragment*, whereas our `JUCQ`s do not have this constraint; further, the `UCQ` and `SCQ` reformulations *do not take into account quantitative information* about the data and query.

We compared the performance of query answering through: (*i*) `UCQ` reformulation; (*ii*) `SCQ` reformulation; (*iii*) the `JUCQ` recommended by the exhaustive ECov algorithm; (*iv*) the `JUCQ` recommended by our greedy GCov algorithm.

Figure 4 shows the evaluation times for LUBM queries on the 100M dataset, on the three RDBMSs we tested; observe the logarithmic time axis. Missing bars correspond to executions which timed out or were infeasible. Figure 4 shows that neither `UCQ` nor `SCQ` reformulation are reliable options. Indeed, `UCQ` *is the slowest* for many queries on System Aand Postgres, sometimes by more than an order of magnitude, and it *fails* for $Q_9, Q_{15}, Q_{18}(forLUBM100M), Q_{19}$ and Q_{28} on System A, to which we add Q_6, Q_{14} and Q_{16} on Postgres (for LUBM 100M). `SCQ` *is very inefficient* on System B, and also on Postgres for Q_1, Q_2, Q_3, Q_8 etc.; it is almost always the worst choice for System B. In contrast, the GCov-chosen `JUCQ` *always completes* and is *the fastest overall* in all but Q_{24}, Q_{25} and Q_{27} on Postgres. Figure 4 also shows that the GCov `JUCQ` performs as well as the ECov one, thus the greedy is making smart choices. In Figure 4, the GCov `JUCQ` is *up to 4 orders of magnitude faster than the* `SCQ` *reformulation* and *two orders of magnitude faster than* `UCQ` (on

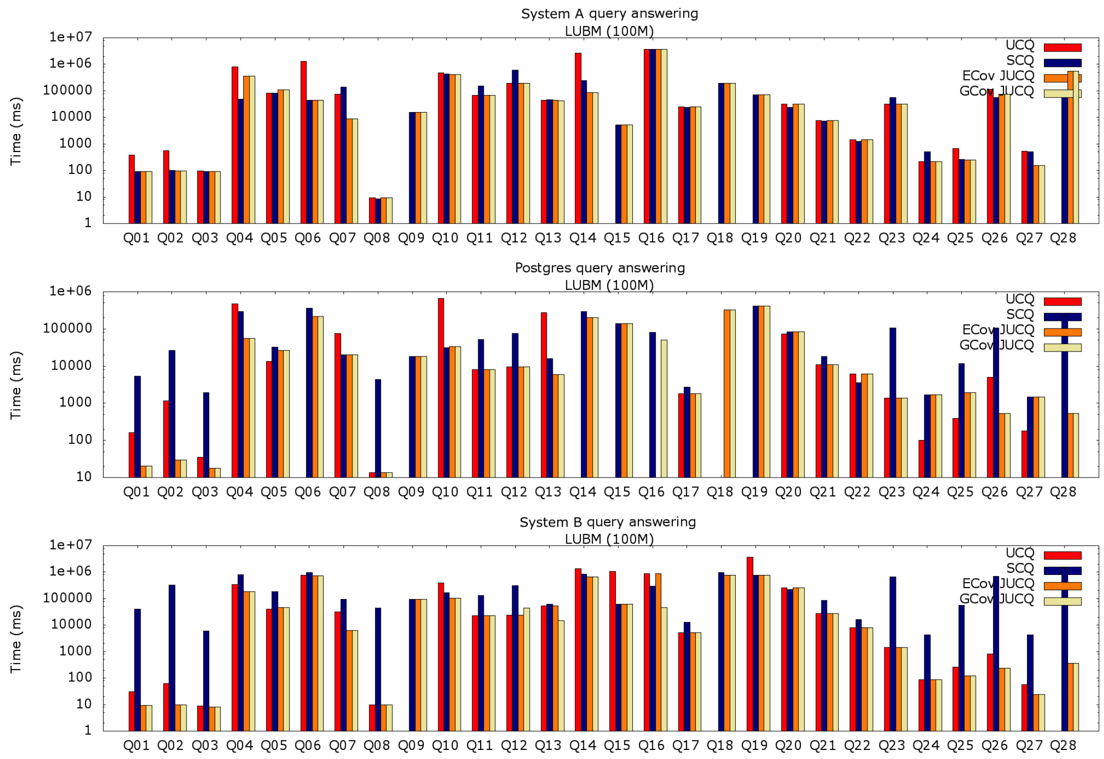


Figure 4: LUBM 100M query answering through `UCQ`, `SCQ`, ECov and GCov `JUCQ` reformulations, against System A, Postgres and System B.

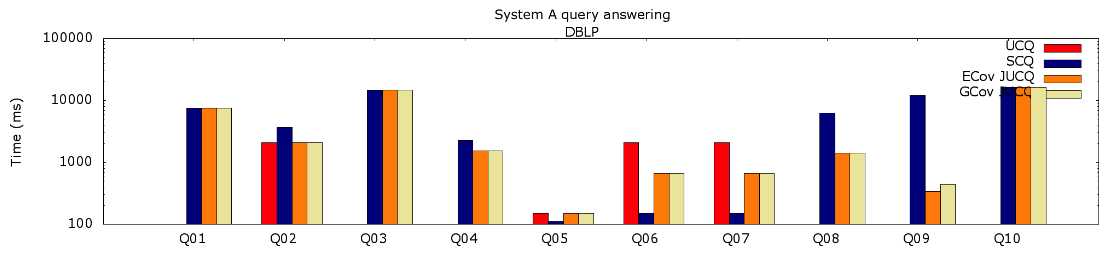


Figure 5: DBLP query answering through `UCQ`, `SCQ`, ECov and GCov `JUCQ` reformulation, against System A.

LUBM 1M, it wins by 3 orders of magnitude w.r.t. `UCQ` [29]). We end by noting that the Q_{16} cover chosen by ECov for Postgres has failed to execute due to insufficient memory in our runtime environment; we believe this could be avoided by further tuning the server execution parameters etc.

Figure 5 further highlights that no fixed reformulation technique is always the best, not even if one fixes the system and the dataset. In this figure, `SCQ` performs very well for Q_6 and Q_7, and very poorly for Q_8 and Q_9; on the latter, `UCQ` times out. In contrast, `JUCQ` performance is robust, the best in all cases but Q_6 and Q_7, and for those it is not very far from the optimum. These experiments highlight the interest of the `JUCQ` reformulation space, and the usefulness of our cost model in guiding ECov and GCov search.

GCov performance We now turn to considering *the number of covers*: overall (as explored by the exhaustive ECov), and the subset traversed by our greedy GCov; these are depicted in Figure 6 also in logarithmic scale. While the search space can be very large (e.g., for LUBM Q_2, Q_9 or Q_{12}), GCov only explores a small subset thereof. The same figure also shows *the running time* of GCov, ECov, and the time to build the `UCQ` and `SCQ` reformulations respectively (again, observe the logarithmic time axis). The time is spent to: obtain the statistics necessary for estimating the number of results of various fragments; reformulate each fragment, estimate its cost, and all other steps shown in Algorithm 1. We see that GCov's running time may be one order of magnitude less than the one of ECov; building the (cost-ignorant) `UCQ` and `SCQ` is faster, but we have seen that their evaluation may be very inefficient. Our algorithms last longest for queries with a huge `UCQ` reformulation (such as LUBM Q_{28}, recall Section 5.1) and/or on queries with *many joins between triples*, such as LUBM Q_{12}: such queries enable many possibilities to add an triple to a fragment, leading to a new cover (recall from Definition 3.3 that a fragment in a cover is not allowed to contain a cartesian product).

Alternative: using the RDBMS cost estimation The second question we study is the quality of our cost estimation, that is crucial in guiding GCov decisions. The golden standard one can compare against is the RDBMS's internal cost estimation function: this is because any cover we chose is evaluated by sending it (as a SQL statement) to the system which optimizes it according to its internal cost model. Thus, the cost function used by GCov should be as close as possible to the RDBMS one.

For this comparison, whenever we needed to estimate the

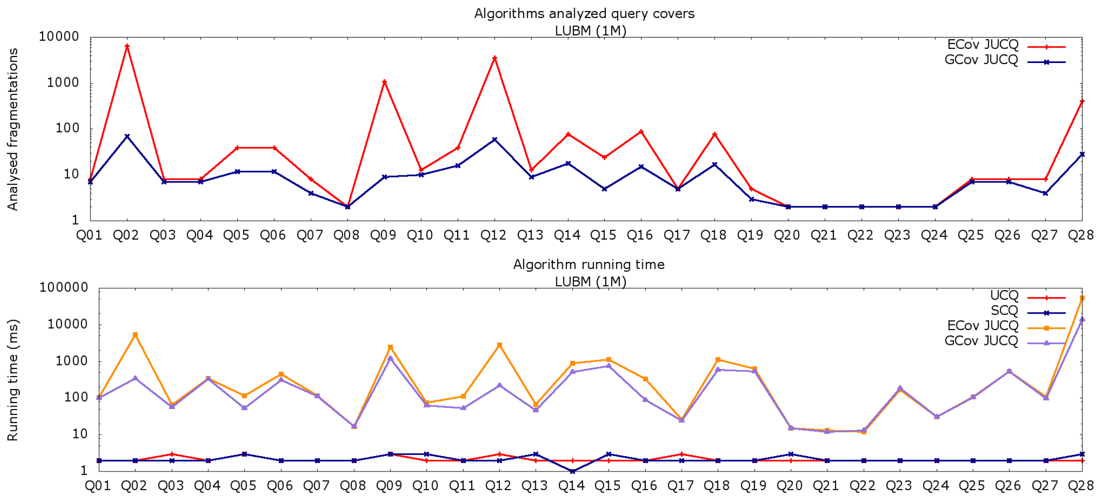


Figure 6: Number of query covers explored by the algorithms (top) and algorithm running times (bottom) for the LUBM queries.

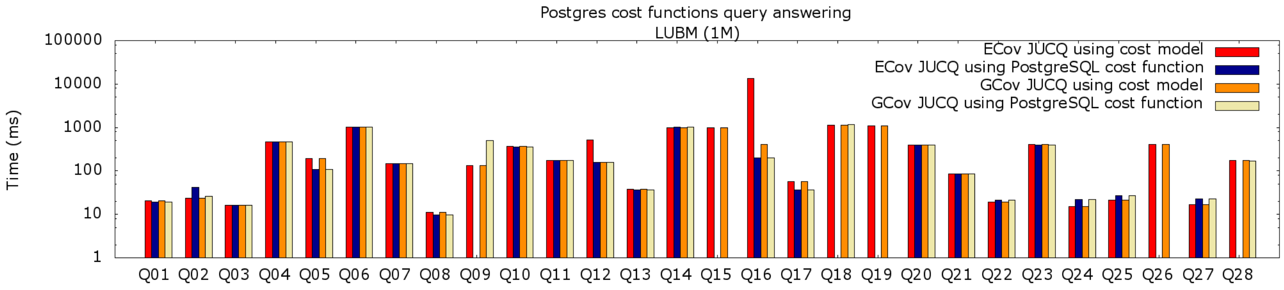


Figure 7: Cost model comparison.

cost of a cover, we sent to Postgres an `explain` statement for the corresponding cover-based reformulation, and extracted from its result Postgres' cost estimation[3]

Figure 7 shows the evaluation time of the `JUCQ` reformulations chosen by ECov and GCov, based on one hand on our cost function, and on the other hand on the Postgres one. Most of the time, the results are similar, demonstrating that our cost model is indeed close to the one of Postgres. In a few cases (LUBM Q_{12} and Q_{16}), using Postgres' cost model helped avoid bad ECov decisions; however, for the LUBM queries Q_9, Q_{15}, Q_{18}, Q_{19}, Q_{26} and Q_{28}, the ECov `JUCQ` chosen based on Postgres' cost estimation was unfeasible.

Figure 7 demonstrates that our cost model (Section 4.1) has lead our algorithm to evaluation choices very similar to the ones that Postgres made, validating its accuracy.

5.3 Comparison with saturation

As explained in the Introduction, graph saturation and query reformulation are the two main techniques for answering queries under constraints. Saturation-based query answering can be very efficient, once the data is saturated; however, if the RDF graph is updated, the cost of maintaining the saturation may be very high [4]. In contrast, query reformulation is performed directly at query time, and so it naturally adapts to the current state of the database. The performance trade-off between saturation- and reformulation-based query answering depends on the schema, on the nature of updates, and on the data statistics [4].

In this section, we show how our optimized `JUCQ` reformulation-based query answering technique impacts the performance comparison with saturation-based query answering. Figure 8 compares on the LUBM 1M dataset: (i) `UCQ` reformulation; (ii) saturation-based query answering based on Postgres; (iii) saturation-based query answering based on Virtuoso v6.1.6 (open-source, multithreaded edition); and (iv) our GCov-chosen `JUCQ`. As expected, `UCQ` reformulation performs much worse than saturation-based query answering, and worse than the GCov `JUCQ` by up to three orders of magnitude. On some queries, such as Q_{15} or $Q_{23} - Q_{28}$, saturation keeps its advantage even compared to our optimized `JUCQ` reformulation. However, on queries such as $Q_3 - Q_{14}$ and $Q_{16} - Q_{22}$, the `JUCQ` reformulation *is close to (competitive with) saturation-based query answering*, which is remarkable given that reformulation reasons at query time, and considering the performance gap observed between the two in previous works, e.g., [4].

5.4 Experiment conclusion

Our experiments lead to the following conclusions. **(1).** Confirming the intuition given by our example in Section 3, the space of `JUCQ` reformulation comprises alternative reformulations of a given BGP query w.r.t. the RDFS constraints, whose evaluation is (i) *feasible when* `UCQ` *reformulation fails*, and (ii) up to *4 orders of magnitude more efficient* than a fixed reformulation strategy, such as `UCQ` or `SCQ`. **(2).** While ECov is slow for large-reformulation queries, GCov identifies covers leading to efficient reformulations quite fast, confirming the feasibility of our optimized reformulation technique at query time. **(3).** The cost model

[3]Doing this for every examined cover slowed down our search significantly, thus we do not recommend actually running GCov *out* of a RDBMS based on the RDBMS's *internal* cost model.

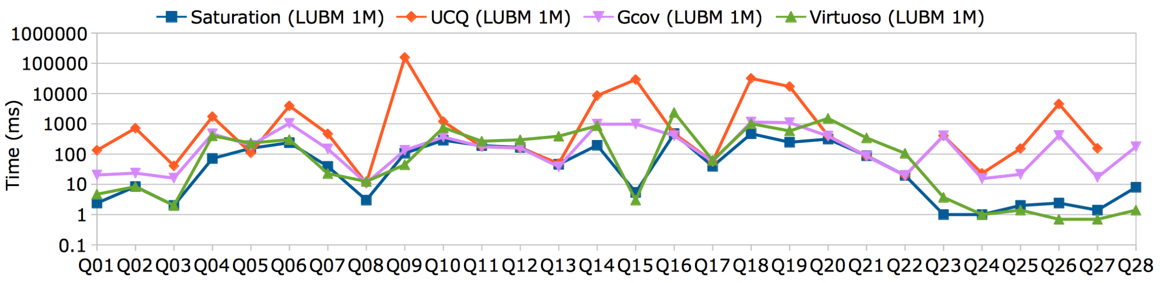


Figure 8: Query answering through Virtuoso and Postgres (via saturation, respectively, optimized reformulation).

on which our search is based performs globally well; in particular, when calibrated for Postgres, we have shown it leads to chosing covers very close to the ones obtained when relying on Postgres' internal cost model. **(4).** While saturation-based query answering has reasons to be much more efficient than reformulation techniques (*if one is willing to disregard the initial cost of saturating the database, as well as any cost related to saturation maintenance!*), our efficient reformulation technique is in many cases competitive with saturation-based query answering, *both through a relational server and through the native-RDF Virtuoso server.* This confirms the important performance improvement brought by our work to reformulation-based query answering in RDF; recall that any CQ to UCQ reformulation algorithm could be used with our cost-based GCov optimization technique.

6. RELATED WORK

The context of our work is the problem of answering conjunctive queries against RDF facts, in the presence of RDFS constraints. As mentioned in the introduction, solutions from the literature rely on RDF graph saturation, on query reformulation, or by mixing both [11]; our work focused on making query answering based on reformulation performant. Below, we position our work w.r.t. these two techniques.

Saturation-based query answering. When using graph saturation, all the implicit triples are computed and explicitly added to the database; query answering then reduces to query evaluation on the saturated database. Well-known SPARQL compliant RDF platforms such as 3store [31], Jena [32], OWLIM [33], Sesame [34], Oracle Semantic Graph [35] support saturation-based query answering, based on (a subset of) RDF entailment rules.

RDF platforms originating in the data management community, such as Hexastore [9] or RDF-3X [8], ignore entailed triples and only provide query *evaluation* on top of the RDF graph, which is assumed to be already saturated.

The drawbacks of saturation w.r.t. updates have been pointed out in [3], which proposes a *truth maintenance* technique implemented in Sesame. It relies on the storage and management of the *justifications* of entailed triples (which triples beget them). This technique incurs a high overhead of handling justifications when their number and size grow. Therefore, [36] proposes to compute only the relevant justifications w.r.t. an update, at maintenance time. This technique is implemented in OWLIM, however [33] points out that updates upon RDFS constraint deletions can lead to poor performance. More efficient saturation maintenance techniques are provided in [4, 6] based on the *number of times* triples are entailed.

Reformulation-based query answering. When using query reformulation, a given BGP query is reformulated based on the RDFS constraints into a target language, such that evaluating the reformulated query through an appropriate engine yields the query answer.

UCQ reformulation [4, 6, 10, 11, 12, 14, 15, 17, 18] applies to various fragments of RDF, ranging from the Description Logics (DL) one up to the Database one, the largest for which this technique have been considered so far. UCQ reformulation corresponds in this work to a JUCQ reformulation obtained from a single fragment query cover. SCQ reformulation [13] was defined for the DL fragment of RDF. In our setting, it corresponds to a JUCQ reformulation obtained from a query cover in which each triple is alone in a fragment. Our experiments have shown that the evaluation performance for both UCQ or SCQ reformulation can be very poor.

Among popular RDF data management systems, the only ones supporting reformulation-based query answering are Stardog, Virtuoso (which supports only the rdfs:subClassOf and rdfs:subPropertyOf RDFS rules) and AllegroGraph [37] which supports the four RDFS rules but whose reasoning implementation is incomplete[4]. Virtuoso is based on SCQ reformulation, while Stardog uses UCQ reformulation; we found no information about AllegroGraph's query reformulation language. Nested SPARQL is the target reformulation language in [19]; in contrast, we focus on translating into a commonly supported language such as JUCQs which in turn can be efficiently evaluated by an SQL engine. In [11], the schema is maintained saturated and reformulation is applied at runtime. Our approach could apply in that setting, to improve their reformulation performance.

Datalog has also been used as a target reformulation language. For instance, Presto [12, 38] reformulates queries in a DL-Lite setting into non-recursive Datalog programs. These DL-Lite formalisms are strictly more expressive from a semantic constraint viewpoint than the RDFS constraints we consider. Thus, their method could be easily transferred (restricted) to the DL fragment of RDF which, as previously mentioned, is a subset of the database fragment of RDF that we consider. However, these works did not consider cost-driven performance optimization based on data statistics and a query evaluation cost model as in our work.

From a *database optimization* perspective, the performance advantage we gain by adding selective triples next to very large ones within query covers' fragments is akin to the semi-join reducers technique, well-known from the distributed database context [39]. It has been shown e.g., in [40] that semi-join reducers can also be beneficial in a centralized context by reducing the overall join effort. In this work, we use a technique reminiscent of semi-joins in order to pick the best *query-level* formulation of a reformulated query, to make its

[4]As stated at http://franz.com/agraph/support/documentation/v4/reasoner-tutorial.html#fnr0-2014-09-16

evaluation possible and efficient; this contrasts with the traditional usage of semi-joins *at the level of algebraic plans*. On one hand, working at the plan level enables one to intelligently combine traditional joins and semi-joins to obtain the best performance. On the other hand, producing (as we do) an output at the *query (syntax)* level (recall Figure 1) enables us to take advantage of any existing system, and of its optimizer which will figure out the best way to evaluate such queries, a task at which many systems are good once the query has a "reasonable" shape and size. Further, expressing optimized reformulations as queries allows us *not* to (re-)explore the search space of join orders etc. together with the (already large) space of possible reformulated queries.

7. CONCLUSION

Our work is placed in the setting of query answering against RDF graphs in the presence of RDF Schema constraints. In particular, we focus on *improving the performance of reformulation-based RDF query answering.*

We have identified a space of alternative JUCQ reformulations, whose evaluation (based on a standard, semantics-unaware query processor) may be (*i*) feasible even when the prominent UCQ reformulation is not, and (*ii*) more efficient by up to three orders of magnitude. Further, we have presented a cost model for such JUCQ alternatives, and proposed an anytime greedy cost-based algorithm capable of identifying such efficient alternatives. Our technique may be used with any CQ-to-UCQ query reformulation algorithm (recall Figure 1) and thus we consider it a big step forward toward making reformulation-based query answering efficient. This is particularly useful in contexts when the data and/or constraints are updated, and saturation-based techniques incur high maintenance costs as illustrated e.g., in [4]; in contrast, applying at query time, reformulation-based query answering is naturally robust to updates, and (through cost-based techniques such as the one described in our work) close to saturation-based performance but without its drawbacks.

Acknowledgements This work has been partially funded by Datalyse.

8. REFERENCES

[1] "Resource description framework." http://www.w3.org/RDF.
[2] "SPARQL protocol and RDF query language." http://www.w3.org/TR/rdf-sparql-query.
[3] J. Broekstra and A. Kampman, "Inferencing and truth maintenance in RDF Schema: Exploring a naive practical approach," in *PSSS Workshop*, 2003.
[4] F. Goasdoué, I. Manolescu, and A. Roatiş, "Efficient query answering against dynamic RDF databases," in *EDBT*, 2013.
[5] J. Urbani, S. Kotoulas, J. Maassen, F. van Harmelen, and H. E. Bal, "WebPIE: A web-scale parallel inference engine using MapReduce," *J. Web Sem.*, vol. 10, pp. 59–75, 2012.
[6] J. Urbani, A. Margara, C. J. H. Jacobs, F. van Harmelen, and H. E. Bal, "Dynamite: Parallel materialization of dynamic RDF data," in *ISWC*, 2013.
[7] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach, "Scalable semantic web data management using vertical partitioning," in *VLDB*, 2007.
[8] T. Neumann and G. Weikum, "The RDF-3X engine for scalable management of RDF data," *VLDBJ*, vol. 19, no. 1, pp. 91–113, 2010.
[9] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: Sextuple indexing for Semantic Web data management," *PVLDB*, 2008.
[10] Z. Kaoudi, I. Miliaraki, and M. Koubarakis, "RDFS reasoning and query answering on DHTs," in *ISWC*, 2008.
[11] J. Urbani, F. van Harmelen, S. Schlobach, and H. Bal, "QueryPIE: Backward reasoning for OWL Horst over very large knowledge bases," in *ISWC*, 2011.
[12] G. D. Giacomo, D. Lembo, M. Lenzerini, A. Poggi, R. Rosati, M. Ruzzi, and D. F. Savo, "MASTRO: A reasoner for effective ontology-based data access," in *ORE*, 2012.
[13] M. Thomazo, "Compact rewriting for existential rules," *IJCAI*, 2013.
[14] P. Adjiman, F. Goasdoué, and M.-C. Rousset, "SomeRDFS in the semantic web," *JODS*, vol. 8, 2007.
[15] F. Goasdoué, K. Karanasos, J. Leblay, and I. Manolescu, "View selection in semantic web databases," *PVLDB*, 2011.
[16] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, eds., *The DL Handbook: Theory, Implem., and Applications*, Cambridge Univ. Press, 2003.
[17] D. Calvanese, G. D. Giacomo, D. Lembo, M. Lenzerini, and R. Rosati, "Tractable reasoning and efficient query answering in description logics: The DL-Lite family," *JAR*, vol. 39, no. 3, 2007.
[18] G. Gottlob, G. Orsi, and A. Pieris, "Ontological queries: Rewriting and optimization," in *ICDE*, 2011. Keynote.
[19] M. Arenas, C. Gutierrez, and J. Pérez, "Foundations of RDF databases," in *Reasoning Web*, 2009.
[20] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases.* Addison-Wesley, 1995.
[21] T. Imielinski and W. Lipski, "Incomplete information in relational databases," *JACM*, vol. 31, no. 4, 1984.
[22] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds, "SPARQL basic graph pattern optimization using selectivity estimation," in *WWW*, 2008.
[23] F. Goasdoué, K. Karanasos, J. Leblay, and I. Manolescu, "View selection in semantic web databases," *PVLDB*, vol. 5, no. 2, 2011.
[24] F. Picalausa, Y. Luo, G. H. Fletcher, J. Hidders, and S. Vansummeren, "A structural approach to indexing triples," in *ESWC*, 2012.
[25] S. Abiteboul, I. Manolescu, P. Rigaux, M.-C. Rousset, and P. Senellart, *Web Data Management.* Cambridge University Press, 2011.
[26] Y. Guo, Z. Pan, and J. Heflin, "LUBM: A benchmark for OWL knowledge base systems," *J. Web Sem.*, vol. 3, no. 2-3, pp. 158–182, 2005.
[27] T. Hearne and C. Wagner, "Minimal covers of finite sets," *Discrete Mathematics*, vol. 5, pp. 247–251, 1973.
[28] R. Ramakrishnan and J. Gehrke, *Database Management Systems.* NY, USA: McGraw-Hill, Inc., 3 ed., 2003.
[29] "Extended version of this work." Available at https://team.inria.fr/oak/?p=2604.
[30] "DBLP." http://kdl.cs.umass.edu/data/dblp/dblp-info.html.
[31] "3store." http://www.aktors.org/technologies/3store.
[32] "Apache jena." http://jena.apache.org.
[33] "Owlim." http://owlim.ontotext.com.
[34] "Sesame." http://www.openrdf.org.
[35] "Oracle semantic graphs." http://docs.oracle.com/cd/E16655_01/appdev.121/e17895/toc.htm, 2014.
[36] B. Bishop, A. Kiryakov, D. Ognyanoff, I. Peikov, Z. Tashev, and R. Velkov, "OWLIM: A family of scalable semantic repositories," *Semantic Web*, vol. 2, no. 1, 2011.
[37] "AllegroGraph RDFStore Web 3.0 Database." http://franz.com/agraph/allegrograph, 2014.
[38] R. Rosati and A. Almatelli, "Improving query answering over DL-Lite ontologies," in *KR*, 2010.
[39] M. T. Özsu and P. Valduriez, *Principles of Distributed Database Systems, Third Edition.* Springer, 2011.
[40] K. Stocker, R. Braumandl, A. Kemper, and D. Kossmann, "Integrating semi-join-reducers into state-of-the-art query processors," in *ICDE*, 2001.

Resolving XML Semantic Ambiguity

Nathalie Charbel
LE2I Lab. UMR-CNRS, University of Bourgogne (UB), 21078 Dijon, France
nathaliecharbel@gmail.com

Joe Tekli
ECE Dept., Lebanese American University (LAU), 36 Byblos, Lebanon
joe.tekli@lau.edu.lb

Richard Chbeir
LIUPPA Lab., University of Pau & Adour Countries (UPPA), 64600 Anglet, France
richard.chbeir@univ-pau.fr

Gilbert Tekli
NOBATEK,
67 Rue de Mirambeau
64600 Anglet, France
gtekli@gmail.com

ABSTRACT

XML semantic-aware processing has become a motivating and important challenge in Web data management, data processing, and information retrieval. While XML data is semi-structured, yet it remains prone to lexical ambiguity, and thus requires dedicated semantic analysis and sense disambiguation processes to assign well-defined meaning to XML elements and attributes. This becomes crucial in an array of applications ranging over semantic-aware query rewriting, semantic document clustering and classification, schema matching, as well as blog analysis and event detection in social networks and tweets. Most existing approaches in this context: i) ignore the problem of identifying ambiguous XML nodes, ii) only partially consider their structural relations/context, iii) use syntactic information in processing XML data regardless of the semantics involved, and iv) are static in adopting fixed disambiguation constraints thus limiting user involvement. In this paper, we provide a new XML Semantic Disambiguation Framework titled *XSDF* designed to address each of the above motivations, taking as input: an XML document and a general purpose semantic network, and then producing as output a semantically augmented XML tree made of unambiguous semantic concepts. Experiments demonstrate the effectiveness of our approach in comparison with alternative methods.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval - Content Analysis and Indexing; I.7.1 [**Document and Text Processing**]: Document and Text Editing – *Document management*; I.7.2 [**Document Preparation**]: Document Preparation – *Markup languages*.

General Terms

Algorithms, Measurement, Performance, Design, Experimentation.

Keywords

XML semantic-aware processing, ambiguity degree, sphere neighborhood, XML context vector, semantic network, semantic disambiguation.

1. INTRODUCTION

In the past decade, there has been extensive research around XML data processing taking advantage of the semi-structured nature of XML documents to improve the quality of Web-based information retrieval and data management applications [28]. The majority of existing approaches use syntactic information in processing XML data, while ignoring the semantics involved [48]. Yet, various studies have highlighted the impact of integrating semantic features in XML-based applications, ranging over semantic-aware query rewriting and expansion [11, 40] (expanding keyword queries by including semantically related terms from XML documents to obtain relevant results), XML document classification and clustering [49, 53] (grouping together documents based on their semantic similarities, rather than performing syntactic-only processing), XML schema matching and integration [13, 55] (considering the semantic meanings and relations between schema elements and data-types), and more recently XML-based semantic blog analysis and event detection in social networks and tweets [2, 7]. Here, a major challenge remains unresolved: *XML semantic disambiguation*, i.e., how to resolve the semantic ambiguities and identify the meanings of terms in XML documents [23], which is central to improving the performance of XML-based applications. The problem is made harder with the volume and diversity of XML data on the Web.

Usually, heterogeneous XML data sources exhibit different ways to annotate similar (or identical) data, where the same real world entity could be described in XML using different structures and/or tagging, depending on the data source at hand (as shown in Figure 1, where two different XML documents describe the same *Hitchcock movie*). The core problem here is lexical ambiguity: a term (e.g., an XML element/attribute tag name or data value) may have multiple meanings (homonymy), it may be implied by other related terms (metonymy), and/or several terms can have the same meaning (synonymy) [23]. For instance (according to a general purpose knowledge base such as WordNet [14]) the term *"Kelly"* in XML document 1 of Figure 1 may refer to *Emmet Kelly: the circus clown*, *Grace Kelly: Princess of Monaco*, or *Gene Kelly: the dancer*. However, looking at its context in the document, a human user can tell that *"Kelly"* here refers to *Grace Kelly*. Yet while seemingly obvious for humans, such semantic ambiguities remain extremely complex to resolve with automated processes.

```
<?xml version= "1.0"?>
<films>
   <picture title= "Rear Window">
      <director> Hitchcock </director>
      <year> 1954 </year>
      <genre> mystery </genre>
      <cast>
         <star> Stewart </star>
         <star> Kelly </star>
      </cast>
      <plot>A wheelchair bound
            photographer spies on his
            neighbors …</plot>
      …
   </picture>
</films>
```

a. Doc 1

```
<?xml version= "1.0"?>
<Movies>
   <Movie year= "1954">
      <Name> Rear Window </Name>
      <Directed_By>Alfred Hitchcock</Directed_By>
      <Actors>
         <Actor>
            <FirstName>Grace</FirstName>
            <LastName>Kelly</LastName>
         </Actor>
         <Actor>
            <FirstName>James</FirstName>
            <LastName>Stewart</LastName>
         </Actor>
      </Actors>
      …
   </Movie>
</Movies>
```

b. Doc 2

Figure 1. Sample documents with different structures and tagging, yet describing the same information.

In this context, *word sense disambiguation* (WSD), i.e., the computational identification of the meaning of words in *context* [39], could be central to automatically resolve the semantic ambiguities and identify the meanings of XML element/attribute

10.5441/002/edbt.2015.25

tag names and data values, in order to effectively process XML documents. While WSD has been widely studied for flat textual data [20, 39], yet, the disambiguation of structured XML data remains largely untouched. The few existing approaches to XML semantic-aware analysis (cf. Section 2) have been directly extended from traditional flat text WSD, and thus show several limitations, motivating this work:

- **Motivation 1:** Completely ignoring the problem of *semantic ambiguity*: most existing approaches perform semantic disambiguation on all XML document nodes (which is time consuming and sometimes needless) rather than only processing those nodes which are most ambiguous,
- **Motivation 2:** Only partially considering the structural relations/context of XML nodes (e.g., solely focusing on parent-node relations [52], or ancestor-descendent relations [50]). For instance, in Figure 1, processing XML node *"cast"* for disambiguation: considering (exclusively) its parent node label (i.e., *"picture"*), its root node path labels (i.e., *"films"* and *"picture"*), or its node sub-tree labels (i.e., *"star"*), remains insufficient for effective disambiguation.
- **Motivation 3:** Making use of syntactic processing techniques such as the *bag-of-words* paradigm [49, 52] (commonly used with flat text) in representing XML data as a plain set of words/nodes, thus neglecting XML structural and/or semantic features as well as compound node labels,
- **Motivation 4:** Existing methods are mostly static in adopting a fixed context size (e.g., parent node [52], or root path [50]) or using preselected semantic similarity measures (e.g., edge-based measure [29], or gloss-based measure [50]), such that user involvement/system adaptability is minimal.

The main goal of our study is to provide an effective method to XML semantic analysis and disambiguation, overcoming the limitations mentioned above. We aim to transform traditional *syntactic XML trees* into *semantic XML trees* (or graphs, when hyperlinks come to play), i.e., XML trees made of concept nodes with explicit semantic meanings. Each concept will represent a unique lexical sense, assigned to one or more XML element/attribute labels and/or data values in the XML document, following the latter's structural context. To do so, we introduce a novel XML Semantic Disambiguation Framework titled *XSDF*, a fully automated solution to semantically augment XML documents using a machine-readable semantic network (e.g., WordNet [14], Roget's thesaurus [60], FOAF [2], etc.), identifying the semantic definitions and relationships among concepts in the underlying XML structure. Different from existing approaches, *XSDF* consists of four main modules for: i) linguistic pre-processing of XML node labels and values to handle compound words (neglected in most existing solutions), ii) selecting ambiguous XML nodes as primary targets for disambiguation using a dedicated *ambiguity degree* measure (unaddressed in existing solutions), iii) representing target nodes as special *sphere neighborhood* vectors considering a comprehensive XML structure context including all XML structural relations within a (user-chosen) range (in contrast with partial context representations using the *bag-of-words* paradigm), and iv) running *sphere neighborhood* vectors through a hybrid disambiguation process, combining two approaches: *concept-based* and *context-based* disambiguation, allowing the user to tune disambiguation parameters following her needs (in contrast with static methods). We have implemented *XSDF* to test and evaluate our approach. Experimental results reflect our approach's effectiveness in comparison with existing solutions.

The remainder of this paper is organized as follows. Section 2 reviews the background and related works. Section 3 develops our XML disambiguation framework. Section 4 presents experimental results. Section 5 concludes the paper with future works.

2. BACKGROUND & RELATED WORKS

2.1 Word Sense Disambiguation

WSD underlines the process of computationally identifying the senses (meanings) of words in context, to discover the author's intended meaning [20]. The general WSD task consists of the following main elements: i) selecting words for disambiguation, ii) identifying and representing word contexts, iii) using reference knowledge sources, and iv) associating senses with words.

Selecting words for disambiguation: two possible methods exist: i) *all-words*, or ii) *lexical-sample*. In all-words WSD, e.g., [10, 44], the system is expected to disambiguate all words in a (flat) textual document. In lexical-sample WSD, e.g., [18, 44], specific target words are selected for disambiguation, which are usually the most ambiguous words, chosen using supervised learning methods trained to identify salient words in phrases [39].

Identifying and representing context: the *context* of a word in traditional flat textual data usually consists of the set of terms in the word's vicinity, i.e., terms occurring to the left and right of the considered word, within a certain predefined window size [26]. Thus, the traditional *bag-of-words* paradigm to represent context terms is broadly adopted with flat textual data [20, 39].

Using reference knowledge sources: distinguished as *corpus-based* or *knowledge-based*. The corpus-based approach, e.g., [3, 4, 11], considers previously disambiguated words, and requires supervised learning from sense-tagged corpora (e.g., SemCor [36]) to enable predictions for new words. Knowledge-based methods, e.g., [33, 39, 49], use machine-readable knowledge bases (i.e., ontologies, thesauri and/or taxonomies, e.g., WordNet [14], Roget's thesaurus [60], ODP [28], etc.) providing ready-made sense inventories to be exploited in WSD.

Associating senses with words: categorized as *supervised* or *unsupervised*. Supervised methods, e.g., [31, 39, 57], use machine-learning techniques with *corpus-based* training data provided to a learning algorithm that induces rules to be used for assigning meanings to words. Unsupervised methods, e.g., [29, 43, 48], are usually *knowledge-based* where reference knowledge bases (e.g., WordNet) are processed as *semantic networks* made of concepts representing word senses, and links connecting concepts, representing semantic relations (hyponymy, meronymy, etc., [14, 46], cf. Figure 2). Here, WSD consists in identifying the semantic concept (word sense) in the semantic network that best matches the semantic concepts of terms appearing in the context of the target word, using a measure of *semantic similarity* [9, 42].

Semantic similarity measures in a semantic network: can be classified as *edge-based*, *node-based*, and *gloss-based* [9]. Edge-based methods [25, 59] estimate similarity as the shortest path (in edges, weights, or number of nodes) between concepts being compared. Node-based approaches [27, 45] estimate similarity as the maximum amount of information content concepts share in common, based on the statistical distribution of concept (term) occurrences in a text corpus (e.g., the Brown corpus [15]). Gloss-based methods [5, 6] evaluate word overlap between the glosses of concepts being compared, a gloss underlining the textual definition describing a concept (e.g., the gloss of the 1st sense of word *Actor* in WordNet is *"A theatrical performer"*, cf. Figure 2).

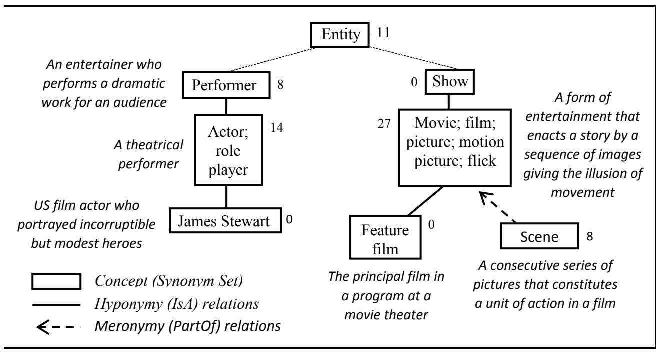

Entity 11
An entertainer who performs a dramatic work for an audience
Performer 8
0 Show
A form of entertainment that enacts a story by a sequence of images giving the illusion of movement
A theatrical performer
Actor; role player 14
27 Movie; film; picture; motion picture; flick
US film actor who portrayed incorruptible but modest heroes
James Stewart 0
Feature film 0
Scene 8
Concept (Synonym Set)
Hyponymy (IsA) relations
Meronymy (PartOf) relations
The principal film in a program at a movie theater
A consecutive series of pictures that constitutes a unit of action in a film

Figure 2. Extract of the (weighted) WordNet semantic network. Numbers next to concepts represent concept frequencies (based on the Brown text corpus [15]). Sentences next to concepts represent concept glosses.

Note that *unsupervised/knowledge-based* WSD has been largely investigated recently (including most methods targeting XML data), in comparison with *supervised* and *corpus-based* methods, which usually require extensive training and large test corpora [39], and thus do not seem practical for the Web. The reader can refer to [20, 39] for reviews on traditional WSD.

2.2 XML Semantic Disambiguation

Few approaches have been developed for semantic disambiguation of XML and semi-structured data. The main challenges reside in the notion of XML (structural) contextualization and how it is processed, as described below.

2.2.1 XML Context Identification

While the context of a word in traditional flat textual data consists of the set of terms in the word's vicinity [26], yet there is no unified definition of the context of a node in an XML document tree. Different approaches have been investigated: i) *parent node*, ii) *root path*, iii) *sub-tree*, and iv) *versatile structural context*.

Parent node context: The authors in [51, 52] consider the *parent node* to be the context of an XML data element, and process a parent node and its children as one (canonical) entity, deemed as the simplest semantically meaningful structural entity. The authors utilize context-driven search techniques (structure pruning, identifying relatives, etc.) to determine the relations between canonical trees. These are used to assign semantic node labels using a reference ontology [47], generalizing/specializing node concepts following their labels and positions in the XML tree.

Root path context: In [49, 50], the authors extend the notion of XML node context to include the whole XML root path, i.e., the path consisting of the sequence of nodes connecting a given node with the root of the XML document (or document collection). They perform per-path sense disambiguation, comparing every node label in each path with all possible senses of node labels occurring in the same path (using gloss-based and edge-based semantic similarity measures from [6, 59] applied on WordNet) to select the appropriate sense for the node label being processed.

Sub-tree context: The authors in [56] consider the set of XML node labels contained in the sub-tree rooted at a given element node to describe the node's XML context. The authors apply a similar paradigm to identify the contexts of all possible node label senses in WordNet. Consequently, they compare the XML label context to all candidate sense contexts in WordNet, identifying the sense (concept) with the highest context similarity.

Versatile structural context: In [29], the authors combine the notions of parent context and descendent (sub-tree) context in disambiguating generic structured data (e.g., XML, web directories, and ontologies). They propose various edge-weighting measures (namely a Gaussian decay function) to identify *crossable* edges, such that nodes reachable from a target node through any *crossable* edge belong to the target node's context. Then, they compare the target node label with each candidate sense (concept) corresponding to the labels in the target node's context (using edge-based semantic similarity [24] applied on WordNet) in order to identify the highest matching concept.

2.2.2 XML Context Representation and Processing

Another concern in XML-based WSD is how to effectively process the context of an XML node (once it has been identified) considering the structural positions of XML data. Most existing WSD methods - developed for flat textual data (Section 2.1) and/or XML-based data [49-52] - adopt the *bag-of-words* model where context is processed as a set of words surrounding the term/label (node) to be disambiguated. Hence, all context nodes are treated the same, despite their structural positions in the XML tree. One approach in [29] extends the traditional *bag-or-words* paradigm with additional information considering distance weights separating the context and target nodes: identified as *relational information model* [29]. The authors employ a specially tailored Gaussian decay function estimating edge weights such as the closer a node (following a user-specified direction), the more it influences the target node's disambiguation [29].

2.2.3 Associating Senses with XML Nodes

Once the contexts of XML nodes have been determined, they can be handled in different ways. Two interesting approaches, both *unsupervised* and *knowledge-based*, have been adopted in this context, which we identify as: *concept-based* and *context-based*. On one hand, the concept-based approach adopted in [49, 50] consists in evaluating the semantic similarity between target node senses (concepts) and those of its context nodes, using measures of semantic similarity between concepts in a semantic network, selecting the target sense with maximum similarity. On the other hand, the context-based approach introduced in [56] consists in building context vectors for each target node sense (concept) in the semantic network, and building a context vector for the target node in the XML document tree, and then comparing context vectors to select the target sense with maximum vector similarity.

A hybrid approach in [29] combines variants of the two preceding approaches to disambiguate generic structured data (including XML). Yet while producing quality results, the authors do not compare their solution with XML disambiguation methods.

Wrapping up: we identify four major limitations motivating our work (which have been highlighted in Section 1): most existing methods i) completely ignore the problem of *semantic ambiguity*, ii) only partially consider the structural relations/context of XML nodes (e.g., parent-node [52] or ancestor-descendent relations [50]), ii) neglect XML structural/semantic features by using syntactic processing techniques such as the *bag-of-words* paradigm [49, 52], and iv) are static in choosing a fixed context (e.g., parent node [52], or root path [50]) or preselected semantic similarity measures, thus minimizing user involvement.

3. XML DISAMBIGUATION FRAMEWORK

In order to address all motivations above and provide a more complete and dynamic XML disambiguation approach, we introduce *XSDF* as an unsupervised and knowledge-based solution to resolve semantic ambiguities in XML documents. *XSDF*'s overall architecture is depicted in Figure 3. It is made of four modules: i) linguistic pre-processing, ii) nodes selection for disambiguation, iii) context definition and representation, and iv) XML semantic disambiguation. The system receives as input: i) an XML document tree, ii) a semantic network (noted *SN*), and iii) user parameters (to tune the disambiguation process following her needs), and produces as output a semantic XML tree.

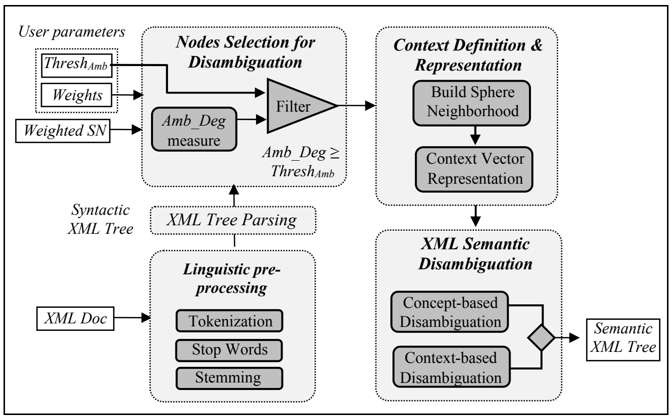

User parameters
$Thresh_{Amb}$
Weights
Weighted SN
Nodes Selection for Disambiguation
Amb_Deg measure
Filter
$Amb\_Deg \geq Thresh_{Amb}$
Context Definition & Representation
Build Sphere Neighborhood
Context Vector Representation
Syntactic XML Tree
XML Tree Parsing
XML Semantic Disambiguation
Linguistic pre-processing
XML Doc
Tokenization
Stop Words
Stemming
Concept-based Disambiguation
Context-based Disambiguation
Semantic XML Tree

Figure 3. Overall *XSDF* architecture.

We develop *XSDF*'s main modules in the following, starting with the XML and semantic data models adopted in our study.

3.1 XML and Semantic Data Models

XML documents represent hierarchically structured information and can be modeled as *rooted ordered labeled trees* (Figure 4.a and b), based on the Document Object Model (DOM) [58].

Definition 1 - Rooted Ordered Labeled Tree: It is a rooted tree in which the nodes are labeled and ordered. We denote by $T[i]$ the i^{th} node of T in preorder traversal, $T[i].\ell$ its label, $T[i].d$ its depth (in number of edges), and $T[i].f$ its out-degree (i.e., the node's fan-out). $R(T)$=[0] designates the root node of tree T[1] ●

An XML document can be represented as a *rooted ordered labeled tree* where nodes represent XML elements/attributes, labeled using element/attribute tag names. Element nodes are ordered following their order of appearance in the XML document. Attribute nodes appear as children of their containing element nodes, sorted[2] by attribute name, and appearing before all sub-elements [41, 61]. Element/attribute text values are stemmed and decomposed into tokens (using our linguistic pre-processing component), mapping each token to a leaf node labeled with the respective token, appearing as a child of its container element/attribute node, and ordered following their order of appearance in the element/attribute text value (Figure 4.a).

Note that element/attribute values can be disregarded (*structure-only*) or considered (*structure-and-content*) in the XML document tree following the application scenario at hand. Here, we believe integrating XML structure and content is beneficiary in resolving ambiguities in both element/attribute tag names (structure) and data values (content). For instance, in the document of Figure 1.a, considering data values *"Kelly"* and *"Stewart"* would be beneficial to disambiguate tag label *"cast"*. The same applies the other way: *"cast"* can help disambiguate *"Kelly"* and *"Stewart"*. Also, we provide the formal definition of a semantic network, as the semantic (knowledge base) data model adopted in our study[3].

Definition 2 – Semantic network: It is made of concepts representing groups of words/expressions designating word senses, and links connecting the concepts designating semantic relations, and can be represented as $SN=(C, L, G, E, R, f, g)$:

- C: set of nodes representing concepts in *SN* (*synsets* as in WordNet [14]),
- L: set of words describing concept labels,
- G: set of glosses describing concept definitions,
- E: set of edges connecting concept nodes, $E \subseteq C \times C$,
- R: set of semantic relations, R = {*Is-A*, *Has-A*, *Part-Of*, *Has-Part*…}, the synonymous words/expressions being integrated in the concepts themselves,
- f: function designating the labels, sets of synonyms, and glosses of concept nodes, f: $C \rightarrow L, L^n, G$ where n designates the number of synonyms per concept,
- g: function designating the labels of edges, g: $E \rightarrow R$.

Note that $c \in SN$ designates a semantic concept with $c.\ell$ its label, $c.syn$ its set of synonymous words, and $c.gloss$ its gloss. We also designate by $\overline{SN}$ a weighted semantic network: augmented with concept frequencies (cf. Figure 2) statistically quantified from a given text corpus (e.g., the Brown corpus [15]) ●

In our current study and tests, we adopt WordNet [14] as a reference semantic network, being a commonly used lexical reference for the English language. Yet, any other knowledge base can be used based on the application scenario, e.g., ODP [28] for describing semantic relations between Web pages, or FOAF [2] to identify relations between persons in social networks.

Note that after disambiguation, target nodes in the XML document tree would consist of semantic concept identifiers extracted from the reference semantic network, where non-target XML nodes remain untouched (cf. Figure 4.b).

3.2 Linguistic Pre-Processing

Linguistic pre-processing consists of three main phases: i) tokenization, ii) stop word removal, and iii) stemming, applied on each of the input XML document's element/attribute tag names and text values, to produce corresponding XML tree node labels. Here, we consider three possible inputs:

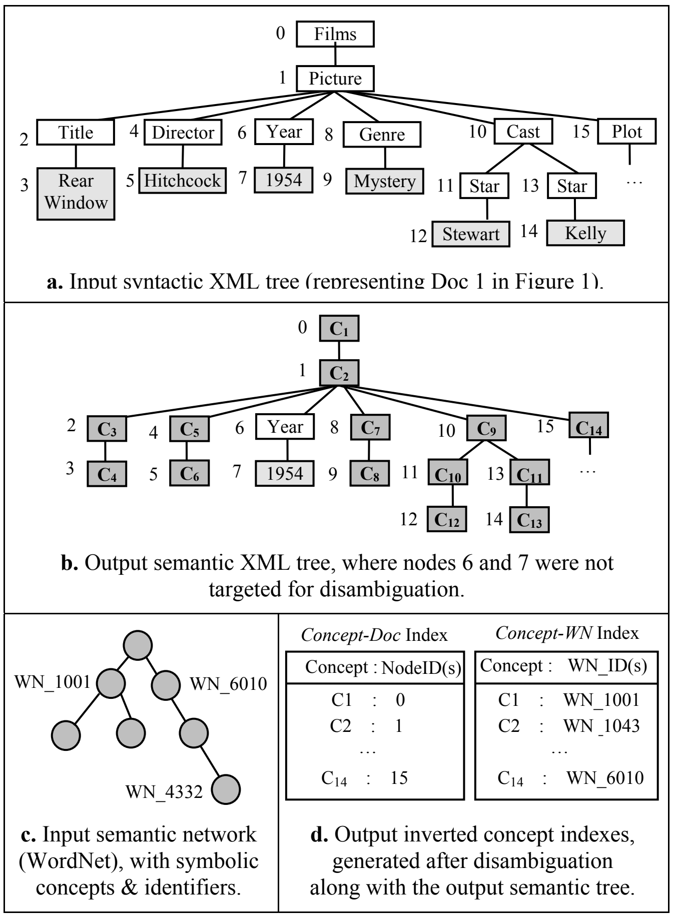
- Element/attribute tag names consisting of individual words,
- Element/attribute tag names consisting of compound words, usually made of two individual terms (t_1 and t_2)[4] separated by special delimiters (namely the underscore character, e.g., "*Directed_By*"), or the use of upper/lower case to distinguish the individual terms (e.g., "*FirstName*"),
- Element/attribute text values consisting of sequences of words separated by the space character.

[1] *Tree* and *rooted ordered labeled tree* are used interchangeably hereafter.

[2] While the order of attributes (unlike elements) is irrelevant in XML [1], yet we adopt an ordered tree model to simplify processing [44, 58].

[3] *Knowledge base* & *semantic network* are used interchangeably hereafter.

[4] More than two terms per XML tag name is unlikely in practice [59].



a. Input syntactic XML tree (representing Doc 1 in Figure 1).

b. Output semantic XML tree, where nodes 6 and 7 were not targeted for disambiguation.

c. Input semantic network (WordNet), with symbolic concepts & identifiers.

d. Output inverted concept indexes, generated after disambiguation along with the output semantic tree.

Figure 4. Sample input (syntactic) XML tree and output (semantic) XML trees.

Considering the first case, no significant pre-processing is required, except for stemming (when the word is not found in the reference semantic network). Considering the second case (i.e., compound words, usually disregarded in existing methods), if t_1 and t_2 match a single concept in the semantic network (i.e., a synset in WordNet, e.g. *first name*), they are considered as a single token. Otherwise, they are considered as two separate terms, and are processed for stop word removal and stemming. Yet, they are kept within a single XML node label (ℓ) in order to be treated together afterward, i.e., one sense will be finally associated to ℓ, which is formed by the best combination of t_1 and t_2's senses (in contrast with studies in [29, 56] which process token senses separately as distinct labels). As for the third case, we apply traditional tokenization (i.e., the text value sentence is broken up into a set of word tokens t_i), processed for stop word removal and stemming, and then represented each as an individual node (x_i) labeled with the corresponding token ($x_i.\ell = t_i$), and appearing as children of the containing element/attribute node.

3.3 Node Selection for Disambiguation

Given an input XML tree, the first step is to select target nodes to disambiguate, which (we naturally assume) are the most ambiguous nodes in the document tree. Thus we aim to provide a mathematical definition to quantify an XML node ambiguity degree which can be used to select target nodes for disambiguation (answering *Motivation 1*). To do so, we start by clarifying our assumptions about XML node ambiguity:

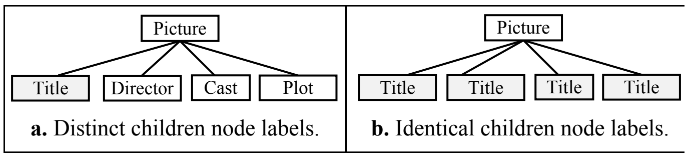
- **Assumption 1:** The semantic ambiguity of an XML node is related to the number of senses of the node's label: i) the more senses it has, the more ambiguous the node is, ii) the less senses it has, the less ambiguous the node is.
- **Assumption 2:** The semantic ambiguity of an XML node is related to its distance from the root node of the document tree: i) the closer it is to the root, the more ambiguous it is, ii) the farther it is from the root, the less ambiguous it is.

Assumption 2 follows the nature of XML and semi-structured data, where nodes closer to the root of the document tend to be more descriptive of the whole document, i.e., having a broader meaning, than information further down the XML hierarchy [8, 61]. In other words, as one descends in the XML tree hierarchy, information becomes increasingly specific, consisting of finer details [54], and thus tends to be less ambiguous.

- **Assumption 3:** The semantic ambiguity of an XML node is related to its number of children nodes having distinct labels: i) the lesser the number of distinct children labels, the more ambiguous the node is, ii) the more the number of distinct children labels, the less ambiguous the node is.

Assumption 3 is highlighted in the sample XML trees in Figure 5. One can clearly identify the meaning of root node label *"Picture"* (i.e., *"motion picture"*) in Figure 5.a., by simply looking at the node's distinct children labels. Yet the meaning of *"Picture"* remains ambiguous in the XML tree of Figure 5.b (having several children nodes but with identical labels). Hence, we believe that distinct children node labels can provide more hints about the meaning of a given XML node, making it less ambiguous.

- **Assumption 4:** An XML node which label has only one possible sense is considered to be unambiguous (i.e., semantic ambiguity is minimal), regardless of its distance from the tree root and its number of distinct children labels.



a. Distinct children node labels.

b. Identical children node labels.

Figure 5. Sample XML document trees.

While our goal is to quantify XML semantic ambiguity, yet this can be done in many alternative ways that would be consistent with the above assumptions. Hence, we first provide a set of propositions that map to the above assumptions, which we will utilize to derive our ambiguity degree measure.

Proposition 1: The ambiguity degree of an XML node x in tree T increases when the number of senses of $x.\ell$ is high in the reference semantic network *SN*, or else it decreases such that:

$$\text{Amb}_{Polysemy}(\text{x}.\ell, \text{SN}) = \frac{senses(\text{x}.\ell) - 1}{\text{Max}(senses(\text{SN})) - 1} \in [0,1] \quad \textbf{(1)}$$

where *Max*(*senses*(*SN*)) is the maximum number of senses of a word/expression in *SN* (e.g., in WordNet 2.1 [14], $Max_{polysemy}$ = 33, for the word *"head"*) □

Proposition 2: The ambiguity degree of an XML node x in tree T increases when the distance in number of edges between x and $R(T)$ is low, or else it decreases such that:

$$\text{Amb}_{Depth}(\text{x, T}) = 1 - \frac{\text{x.d}}{\text{Max}(depth(\text{T}))} \in [0,1] \quad \textbf{(2)}$$

where *Max*(*depth*(*T*)) is the maximum depth in *T* □

Proposition 3: The ambiguity degree of an XML node *x* in tree *T* increases when the number of children nodes of *x* having distinct labels, designated as $\overline{x.f}$, is low, or else it decreases:

$$\text{Amb}_{Density}(\text{x, T}) = 1 - \frac{\overline{\text{x.f}}}{\text{Max}(\overline{fan\text{-}out(\text{T})})} \in [0,1] \quad \textbf{(3)}$$

where $Max(\overline{fan\text{-}out(\text{T})})$ is the maximum number of children nodes with distinct node labels in *T*. We identify this factor as node density factor to distinguish it from traditional node fan-out: number of children nodes (regardless of label, cf. Definition 1) □

From the above propositions, we can derive a general definition for XML ambiguity degree:

Definition 3 – XML Node Ambiguity Degree: Given an XML tree *T*, a node *x* ∊ *T*, and a reference semantic network *SN*, we define the ambiguity degree of *x*, *Amb_Deg*(*x*), as the ratio between $Amb_{Polysemy}(x.\ell, SN)$ on one hand, and the sum of 1-$Amb_{Depth}(x, T)$ and 1-$Amb_{Density}(x, T)$ on the other hand:

$$\text{Amb\_Deg(x, T, SN)} = \frac{\text{w}_{\text{Polysemy}} \times \text{Amb}_{Polysemy}(\text{x}.\ell, \text{SN})}{\text{w}_{\text{Depth}} \times (1-\text{Amb}_{Depth}(\text{x, T})) + \text{w}_{\text{Density}} \times (1-\text{Amb}_{Density}(\text{x, T})) + 1} \in [0,1] \quad \textbf{(4)}$$

where $w_{Polysemy}$, w_{Depth}, $w_{Density}$ ∊ [0, 1] are independent weight parameters allowing the user to fine-tune the contributions of polysemy, depth, and density factors respectively ●

Lemma 1: The ambiguity degree measure *Amb_Deg* in Definition 3 varies in accordance with Propositions 1-3, and conforms to Assumptions 1-4 □

Proofs of Propositions 1-3 and Lemma 1 have been omitted for space limitations, and can be found in [38].

Special case: When the label of node *x* consists of a compound word made of tokens t_1 and t_2, we compute *Amb_Deg*(*x*) as the average of the ambiguity degrees of t_1 and t_2.

Amb_Deg is computed for all nodes in the input XML tree. Then, only the most ambiguous nodes are selected as targets for disambiguation following an ambiguity threshold $Thresh_{Amb}$ automatically estimated or set by the user, i.e., nodes having $Amb\_Deg(x, T, SN) \geq Thresh_{Amb}$, whereas remaining nodes are left untouched. Note that the user can disregard the ambiguity degree measure: i) by setting $w_{Polysemy} = 0$ so that all nodes end up having *Amb_Deg* = 0 regardless of constituent polysemy, depth, and density factors, or ii) by setting $Thresh_{Amb} = 0$ so that all nodes are selected for disambiguation regardless of their ambiguity degrees.

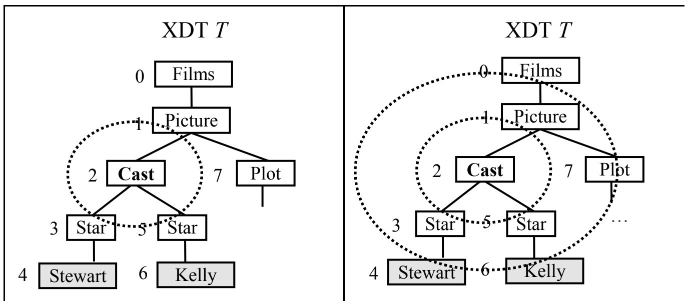
Note that the fine-tuning of parameters is an optimization problem such that parameters should be chosen to maximize disambiguation quality (through some cost function such as *f-measure*, cf. Section 4). This can be solved using a number of known techniques that apply linear programming and/or machine learning in order to identify the best weights for a given problem class, e.g., [19, 30, 37]. Providing such a capability, in addition to manual tuning, would enable the user to start from a sensible choice of values (e.g., identical weight parameters to consider all ambiguity features equally, i.e., $w_{Polysemy} = w_{\text{Depth}} = w_{\text{Fan-out}} = 1$, with a minimal threshold $Thresh_{Amb} = 0$ to consider all results initially) and then optimize and adapt the disambiguation process following the scenario and optimization (cost) function at hand. We do not further address the fine-tuning of parameters here since it is out of the scope of this paper (to be addressed in an upcoming study).

3.4 Context Definition and Representation

3.4.1 XML Sphere Neighborhood

For each target node selected from the previous phase, node contexts have to be defined and processed for disambiguation. While current approaches only partly consider the semi-structured nature of XML in defining disambiguation contexts (*Motivation 2*), we introduce the XML *sphere neighborhood* context model, inspired from the sphere-search paradigm in XML IR [17][5], taking into account the whole structural surrounding of an XML target node (including its ancestors, descendants, and siblings) in defining its disambiguation context. We define the notion of XML ring as the set of nodes situated at a specific distance from the target node. An XML sphere would encompass all rings included at distances less or equal to the size (radius) of the sphere.

Definition 4 – XML Ring: Given an XML tree *T* and a target node *x* ∊ *T*, we define an XML ring with center *x* and radius *d* as the set of nodes located at distance *d* from *x*, i.e., $R_d(x) = \{x_i \in T \mid Dist(x, x_i) = d\}$ ●



a. XML ring $R_1(T[2])$ centered around node *T*[2] of label *"cast"*.

b. XML sphere $S_2(T[2])$ centered around node *T*[2] of label *"cast"*.

Figure 6. Sample XML (ring and) sphere neighborhoods.

The distance between two XML nodes in an XML tree, $Dist(x_i, x_j)$, is typically evaluated as the number of edges separating the nodes. For instance, in tree *T* of Figure 6.a, the distance between nodes *T*[2] and *T*[6] of labels "*cast"* and *"Kelly"* respectively is equal to 2. Hence, the XML ring $R_1(T[2])$ centered around node *T*[2] (*"cast"*) at distance 1 consists of nodes: *T*[1] (*"Picture"*), *T*[3] (*"star"*) and *T*[5] (*"star"*). Note that our approach can be straightforwardly extended to consider different kinds of tree node distance functions (including edge weights, density, or direction, etc. [16, 21]). Yet, we restrict ourselves to the most intuitive notion of node distance here, and report the investigation of other distance functions to a dedicated study.

[5] While comparable with the concept of *XML sphere* exploited in [19], the latter consists of an XML retrieval paradigm for computing TF-IDF scores to rank XML query answers, which is orthogonally different, in its use and objectives, from our disambiguation proposal.

Definition 5 – XML Sphere[6]**:** Given an XML tree *T*, a target node $x \in T$, and a set of XML rings $R_{d_j}(x) \subseteq T$, we define an XML sphere with center *x* and radius *d* as the set of nodes in the rings centered around *x* at distances less or equal to *d*, i.e., $S_d(x) = \{x_i \in T \mid x_i \in R_{d_j}(x) \wedge d_j \leq d\}$ ●

In Figure 6.b, the XML sphere $S_2(T[2])$ centered around node *T*[2] of label *"cast"* with radius 2 consists of: ring $R_1(T[2])$ of radius 1 comprising nodes *T*[1] (*"picture"*), *T*[3] (*"star"*) and *T*[5] (*"star"*), and ring $R_2(T[2])$ of radius 2 comprising nodes *T*[0] (*"Films"*), T[4] (*"Stewart"*), T[6] (*"Kelly"*), and *T*[7] (*"Plot"*). The size (radius) of the XML sphere context is tuned following user preferences and/or the nature of the XML data at hand (e.g., XML trees might contain specialized and domain-specific data, and thus would only require small contexts to achieve good disambiguation, whereas more generic XML data might require larger contexts to better describe the intended meaning of node labels and values, cf. experiments in Section 4).

3.4.2 Context Vector Representation

Having identified the context of a given XML target node, we need to evaluate the impact of each of the corresponding context nodes in performing semantic disambiguation (in contrast with existing methods using the *bag-of-words* paradigm where context is processed as a set of words/nodes disregarding XML structure: *Motivation 3*). Here, we introduce a *relational information* solution based on the general vector space model in information retrieval [32] (in comparison with the specific decay function used in [29]), designed to consider the structural proximity/relations among XML nodes in computing disambiguation scores following our sphere neighborhood model. Our mathematical formulation follows two basic assumptions:

- **Assumption 5:** XML context nodes closer to the target node should better influence the latter's disambiguation, whereas those farther away from the target node should have a smaller impact on the disambiguation process.

This is based on the structured nature of XML, such as nodes closer together in the XML hierarchy are typically more related than more separated ones.

- **Assumption 6:** Nodes with identical labels, occurring multiple times in the context of a target node, should better influence the latter's disambiguation in comparison with nodes with identical labels occurring a lesser number of times.

This is based on the notion of context in WSD, where words occurring multiple times in the context of a target word have a higher impact on the target's meaning. Therefore, we represent the context of a target XML node *x* as a weighed vector, which dimensions correspond to all distinct node labels in its sphere neighborhood context, weighted following their structural distances from the target node.

Definition 6 – XML Context Vector: Given a target node $x \in$ XML tree *T*, and its sphere neighborhood $S_d(x) \in T$, the corresponding context vector $\overrightarrow{V_d(x)}$ is defined in a space which dimensions represent, each, a single node label $\ell_r \in S_d(x)$, such as $1 < r < n$ where *n* is the number of distinct node labels in $S_d(x)$. The coordinate of a context vector $\overrightarrow{V_d(x)}$ on dimension ℓ_r, $w_{\overrightarrow{V_d(x)}}(\ell_r)$, stands for the weight of label ℓ_r in sphere $S_d(x)$ ●

Definition 7 – XML Node Label Weight: The weight $w_{\overrightarrow{V_d(x)}}(\ell_r)$ of node label ℓ_r in context vector $\overrightarrow{V_d(x)}$ corresponding to the sphere neighborhood $S_d(x)$ of target node *x* and radius *d*, consists of the structural frequency of nodes $x_i \in S_d(x)$ having label $x_i.\ell = \ell_r$. It is composed of a normalized occurrence frequency factor $Freq(\ell_r, S_d(x))$ (based on Assumption 6) defined using a structural proximity factor $Struct(x_i, S_d(x))$ (based on Assumption 5). Formally, given $|S_d(x)|$ the cardinality (in number of nodes) of $S_d(x)$:

$$w_{\overrightarrow{V_d(x)}}(\ell_r) = \frac{\text{Freq}(\ell_r, \text{S}_\text{d}(\text{x}))}{\text{Max}_\text{Freq}} = \frac{2 \times \text{Freq}(\ell_r, \text{S}_\text{d}(\text{x}))}{|\text{S}_\text{d}(\text{x})| + 1} \in [0,1] \quad \textbf{(5)}$$

$Freq(\ell_r, S_d(x))$ underlines the total number of occurrences of nodes $x_i \in S_d(x)$ having label $x_i.\ell = \ell_r$, weighted w.r.t. structural proximity, formally:

$$\text{Freq}(\ell_r, \text{S}_\text{d}(\text{x})) = \sum_{\substack{x_i \in S_d(x) \,/ \\ x_i.\ell = \ell_r}} \text{Struct}(\text{x}_\text{i}, \text{S}_\text{d}(\text{x})) \quad \in \left[\frac{1}{d+1}, \frac{|\text{S}_\text{d}(\text{x})|+1}{2}\right] \quad \textbf{(6)}$$

$Struct(x_i, S_d(x))$ underlines the structural proximity between each context node $x_i \in S_d(x)$ having $x_i.\ell = \ell_r$, and the target (sphere center) node *x*, formally:

$$\text{Struct}(\text{x}_\text{i}, \text{S}_\text{d}(\text{x})) = 1 - \frac{\text{Dist}(\text{x}, \text{x}_\text{i})}{\text{d}+1} \quad \in \left[\frac{1}{\text{d}+1}, 1\right] \quad \textbf{(7)}$$

The denominator in $Struct(x_i, S_d(x))$ is incremented by 1 (i.e., *d*+1) to allow context nodes occurring in the farthest ring of the sphere context $S_d(x)$, i.e., the ring $R_d(x)$ of radius *d*, to have a non-null weight in $\overrightarrow{V_d(x)}$, and thus a non-null impact on the disambiguation of target node *x* ●

For instance, given the XML tree in Figure 6, Figure 7 shows context vectors of sphere neighborhoods $S_1(T[2])$ and $S_2(T[2])$ centered around node *T*[2] of label *"Cast"*. One can realize that label weights in Figure 7 increase as nodes occur closer to the target node (Assumption 5), and as the number of node label occurrences increases in the sphere context (Assumption 6, e.g., in $\overrightarrow{V_1(T[2])}$, $w_{\overrightarrow{V_1(\text{“}Cast\text{”})}}(\text{“}Star\text{”}) = 2\times\ w_{\overrightarrow{V_1(\text{“}Cast\text{”})}}(\text{“}Picture\text{”})$ since node label "*Star*" occurs twice in $S_1(T[2])$ while "*Picture*" occurs once; also in $\overrightarrow{V_2(T[2])}$). Formally:

Lemma 2: The context vector weight measure $w_{\overrightarrow{V_d(x)}}(\ell_r)$ in Definition 7 varies in accordance with Assumptions 5 and 6 □

The lemma's proof is omitted here, and can be found in [38], along with detailed computation examples.

In short, context nodes are weighted based on their labels' occurrences as well as the sizes (radiuses) of the sphere contexts to which they correspond, varying context node weights and thus their impact on the target node's disambiguation accordingly.

[6] The notion of *sphere* here is equivalent to that of a *disk* in 2D space. Yet, we adopt the *sphere* nomination for clearness of presentation.

	Cast	*Picture*	*Star*
$\overrightarrow{V_1}(T[2])$	0.4	0.2	0.4

	Cast	*Picture*	*Star*	*Films*	*Stewart*	*Kelly*	*Plot*
$\overrightarrow{V_2}(T[2])$	0.25	0.1667	0.3334	0.0835	0.0835	0.0835	0.0835

Figure 7. Sample sphere context vectors based on the sphere neighborhoods in Figure 6.

3.5 XML Semantic Disambiguation

Once the contexts of XML nodes have been determined, we process each target node and its context nodes for semantic disambiguation. Here, we propose to combine two strategies: the *concept-based* approach and the *context-based* approach. The former is based on semantic concept comparison between target node senses (concepts) and those of its sphere neighborhood context nodes, whereas the latter is based on context vector comparison between the target node's sphere context vector in the XML tree and context vectors corresponding to each of its senses in the reference semantic network. The user will be able to combine and fine-tune both approaches (answering *Motivation 4*).

3.5.1 Concept-based Semantic Disambiguation

It consists in comparing the target node with its context nodes, using a combination of semantic similarity measures (*edge-based*, *node-based*, and *gloss-based*, cf. Section 2.1) in order to compare corresponding semantic concepts in the reference semantic network. Then, the target node sense with the maximum similarity (relatedness) score, w.r.t. context node senses, is chosen as the proper target node sense. To do so, we propose an extension of context-based WSD techniques (cf. Section 2.2.3) where we:

- Build upon the sphere neighborhood context model, to consider XML structural proximity in evaluating the semantic meanings of context nodes (in comparison with the traditional *bag-of-words* context model),
- Allow an extensible combination of several semantic similarity measures, in order to capture semantic relatedness from different perspectives (in comparison with most existing methods which exploit pre-selected measures).

Definition 8 – Concept-based Semantic Score: Given a target node $x \in$ XML tree T and its sphere neighborhood $S_d(x) \in T$, and given s_p as one possible sense for x.ℓ in a (weighted) reference semantic network $\overline{SN}$, we define *Concept_Score*(s_p, $S_d(x)$, $\overline{SN}$) to quantify the semantic impact of s_p as the potential candidate for the intended sense (meaning) of $x.\ell$ within context $S_d(x)$ in T w.r.t. $\overline{SN}$, computed as the average of the weighted maximum similarities between s_p and context node senses:

$$\text{Concept\_Score}(s_p, S_d(x), \overline{SN}) = \sum_{x_i \in S_d(x)} \frac{\underset{s_j^i \to x_i.\ell}{\text{Max}}\left(\text{Sim}(s_p, s_j^i, \overline{SN})\right) \times w_{\overrightarrow{V_d(x)}}(x_i.\ell)}{|S_d(x)|} \in [0,1] \quad \textbf{(8)}$$

where s_j^i designates the *j*th sense of context node $x_i.\ell \in S_d(x)$, and $Sim(s_p, s_j^i, \overline{SN})$ designates the semantic similarity measure between senses s_p and s_j^i w.r.t. $\overline{SN}$ ●

Definition 9 – Semantic Similarity Measure: It quantifies the semantic similarity (relatedness) between two concepts (i.e., word senses) c_1 and c_2 in a reference (weighted) semantic network $\overline{SN}$[7], computed as the weighted sum of several semantic similarity measures[8]. Formally:

$$\begin{aligned}\text{Sim}(c_1, c_2, \overline{SN}) = {} & w_{Edge} \times \text{Sim}_{Edge}(c_1, c_2, SN) + \\ & w_{Node} \times \text{Sim}_{Node}(c_1, c_2, \overline{SN})) + \\ & w_{Gloss} \times \text{Sim}_{Gloss}(c_1, c_2, SN)) \in [0, 1]\end{aligned} \quad \textbf{(9)}$$

where:

- $w_{Edge} + w_{Node} + w_{Gloss} = 1$ and $(w_{Edge}, w_{Node}, w_{Gloss}) \geq 0$
- Sim_{Edge} is a typical edge-based measure from [59],
- Sim_{Node} is a typical node-based measure from [27],
- Sim_{Gloss} is a normalized extension of a typical gloss-based measure from [6] ●

Special case: If the target node label $x.\ell$ is a compound word consisting of two tokens t_1 and t_2 for which no single match was found in the reference semantic network $\overline{SN}$ (cf. Section 3.2), an average score for each possible combination of senses (s_p, s_q) corresponding to each of the individual token senses (s_p for token t_1, and s_q for t_2) is computed to identify the sense combination which is most suitable for the compound target node label:

$$\text{Concept\_Score}((s_p, s_q), S_d(x), \overline{SN}) = \sum_{x_i \in S_d(x)} \frac{\underset{s_j^i \to x_i.\ell}{\text{Max}}\left(\text{Sim}((s_p, s_p), s_j^i, \overline{SN})\right) \times w_{\overrightarrow{V_d(x)}}(x_i.\ell)}{|S_d(x)|} \in [0,1] \quad \textbf{(10)}$$

where

$$\text{Sim}((s_p, s_p), s_j^i, \overline{SN}) = \frac{\text{Sim}(s_p, s_j^i, \overline{SN}) + \text{Sim}(s_q, s_j^i, \overline{SN})}{2} \in [0,1]$$

Note that a compound context node label $x_i.\ell$ which tokens t_1^i and t_2^i do not match any single concept in $\overline{SN}$, is processed similarly to a compound target node label.

3.5.2 Context-based Semantic Disambiguation

It consists in comparing the target node sphere neighborhood in the XML tree with each of its possible sense (concept) sphere neighborhoods in the reference semantic network. To do so, we adopt the same notions of sphere neighborhood and context vector (Definitions 4-7) defined for XML nodes in an XML tree, to build the sphere neighborhood and context vector of a semantic concept in the semantic network. The only difference here is that sphere rings in the semantic network are built using the different kinds of semantic relations connecting semantic concepts (e.g., hypernyms, hyponyms, meronyms, holonyms, cf. Definition 2), in contrast with sphere rings in an XML tree which are built using XML structural containment relations (Definition 1). Here, given

7 $\overline{SN}$ designates a semantic network *SN* weighted with corpus statistics needed to compute a *node*-based similarity measure [29, 48]. Yet, the original non-weighted semantic network *SN* is sufficient to compute typical *edge*-based and *gloss*-based measures (cf. Section 2.1).

8 Here, we use three typical semantic similarity measures, yet any other semantic similarity measure can be used, or combined with the latter.

a reference semantic network *SN*, a semantic concept $c \in SN$, and a radius d, we designate by $R_d(c)$, $S_d(c)$, and $\overrightarrow{V_d(c)}$ the ring, sphere, and context vector of radius d corresponding to concept c in *SN* respectively. Note that linguistic pre-processing (cf. Section 3.2) can be applied to concept labels (when needed[9]) before building context vectors and computing vector weights. Formally:

Definition 10 – Context-based Semantic Score: Given a target node $x \in$ XML tree T, its sphere neighborhood $S_d(x) \in T$ and context vector $\overrightarrow{V_d(x)}$, and given s_p as one possible sense for $x.\ell$ in a reference semantic network *SN*, with its sphere neighborhood $S_d(s_p) \in SN$ and context vector $\overrightarrow{V_d(s_p)}$, we define *Context_Score*(s_p, $S_d(x)$, *SN*) to quantify the semantic impact of s_p as the potential candidate designating the intended sense (meaning) of $x.\ell$ within context $S_d(x)$ in T w.r.t. *SN*, computed using a vector similarity measure between $\overrightarrow{V_d(x)}$ and $\overrightarrow{V_d(s_p)}$:

$$\text{Context\_Score}(s_p, S_d(x), SN) = \cos(\overrightarrow{V_d(x)}, \overrightarrow{V_d(s_p)}) \in [0, 1] \quad \textbf{(11)}$$

where *cos* designates the *cosine* vector similarity measure[10] ●

<u>Special case</u>: If the target node label $x.\ell$ is a compound word consisting of tokens t_1 and t_2 for which no single match was found in the reference semantic network *SN*, an integrated score for each possible combination of senses (s_p, s_q) corresponding to each of the individual token senses (s_p for token t_1, and s_q for token t_2) is computed. Here, the sphere neighborhoods and context vectors of individual senses s_p and s_q are combined together to represent the context sphere of the combination of senses (s_p, s_q) in *SN*:

$$\text{Concept\_Score}((s_p, s_q), S_d(x), SN) = \cos(\overrightarrow{V_d(x)}, \overrightarrow{V_d(s_p, s_q)}) \in [0,1] \quad \textbf{(12)}$$

where $\overrightarrow{V_d(s_p, s_q)}$ is a compound context vector generated based on compound sphere neighborhood $S_d(s_p, s_q) = S_d(s_p) \bigcup S_d(s_q)$.

3.5.3 Combined Semantic Disambiguation

While *concept-based* and *context-based* disambiguation can be applied separately as described in the above sections, yet we allow the user to combine and fine-tune both approaches (answering *Motivation 4*), producing a combined score as the weighted sum of concept-based and context-based scores:

$$\begin{aligned} &\text{Concept\_Score}(s_p, S_d(x), \overline{SN}) = \\ &w_{Concept} \times \text{Concept\_Score}(s_p, S_d(x), \overline{SN}) + \\ &w_{Context} \times \text{Context\_Score}(s_p, S_d(x), SN) \quad \in [0, 1] \end{aligned} \quad \textbf{(13)}$$

where $w_{Concept} + w_{Context} = 1$ and $(w_{Concept}, w_{Context}) \geq 0$

Note that disambiguation algorithms have been omitted for space limitations. Overall complexity simplifies to the sum of the complexities of *concept-based* and *context-based* disambiguation processes, i.e., $O(|senses(x.\ell)| \times |S_d(x)| \times |senses(x_i.\ell)|)$, and $O(|senses(x.\ell)| \times (|S_d(x)| + |S_d(s_p)|)$ respectively (cf. details in [38]).

[9] This depends on the semantic network (not required with WorldNet).

4. EXPERIMENTAL EVALUATION

We have developed a prototype titled *XSDF*[11] to test and compare our approach with its most recent alternatives. We have evaluated two criteria: i) *semantic ambiguity* and ii) *disambiguation quality*.

4.1 Experimental Test Data

We used a collection of 80 test documents gathered from several data sources having different properties (cf. Table 3), which we describe and organize based on two features: i) *node ambiguity*, and ii) *node structure* (cf. Table 1). The former feature highlights the average amount of ambiguity of XML nodes in the XML tree, estimated using our *ambiguity degree* measure, $Amb\_Deg \in [0, 1]$. The latter feature describes the average amount of structural richness of XML nodes, in terms of node *depth*, *fan-out*, and *density* in the XML tree, estimated as the sum of normalized node depth (1-Amb_{Depth}), fan-out, and density (1-$Amb_{Density}$) factors, averaged over all nodes in the XML tree, formally:

$$\text{Struct\_Deg}(x, T) = \frac{w_{Depth} \times x.d}{\text{Max}(\mathit{depth}(T))} + \frac{w_{Fan\text{-}out} \times x.f}{\text{Max}(\mathit{fan\text{-}out}(T))} + \frac{w_{Density} \times \overline{x.f}}{\text{Max}(\overline{\mathit{fan\text{-}out}(T)})} \in [0,1] \quad \textbf{(14)}$$

where $w_{Depth} + w_{Fan\text{-}out} + w_{Density} = 1$ and $(w_{Depth}, w_{Fan\text{-}out}, w_{Density}) \geq 0$. In other words, high node depth, fan-out, and/or density here indicate a highly structured XML tree, whereas low node depth, fan-out, and/or density indicate a poorly structured (relatively flat) tree. In our experiments, we set equal weights $w_{Depth} = w_{Fan\text{-}out} = w_{Density} = 1/3$ when measuring *Struct_Deg* (cf. Table 1). In this study, we do not address the issue of assigning weights, which could be performed using optimization techniques (e.g., linear programming and/or machine learning [19, 30, 37]) to help fine-tune input parameters and obtain optimal results (cf. Section 3.3). Such a study would require a thorough analysis of the relative effect of each parameter on disambiguation quality, which we report to a dedicated subsequent study.

Table 1. XML test documents organized based on average node *ambiguity* and *structure*.

	Structure +	Structure –
Ambiguity +	**Group 1** *Amb_Deg* = 0.1127 & *Struct_Deg* = 0.6803	**Group 2** *Amb_Deg* = 0.1378 & *Struct_Deg* = 0.6621
Ambiguity –	**Group 3** *Amb_Deg* = 0.0625 & *Struct_Deg* = 0.612	**Group 4** *Amb_Deg* = 0.0447 & *Struct_Deg* = 0.5515

4.2 XML Ambiguity Degree Correlation

We compare XML node ambiguity ratings produced by human users with those produced by our system (i.e., via our *ambiguity degree* measure, *Amb_Deg*, cf. Section 3.3), using *Pearson's correlation coefficient*, $pcc = \delta_{XY}/(\sigma_X+\sigma_Y)$ where: x and y designate user and system generated ambiguity degree ratings respectively, σ_X and σ_Y denote the standard deviations of x and y respectively, and δ_{XY} denotes the covariance between the x and y variables. The values of $pcc \in [-1, 1]$ such that: -1 designates that one of the variables is a decreasing function of the other variable (i.e., values deemed ambiguous by human users are deemed unambiguous by the system, and visa-versa), 1 designates that one of the variables is an increasing function of the other variable

[10] We adopt *cosine* since it is widely used in IR [35]. Yet, other vector similarity measures can be used, e.g., Jaccard, Pearson corr. coeff., etc.

[11] Available online at http://sigappfr.acm.org/Projects/XSDF/

(i.e., values are deemed ambiguous/unambiguous by human users and the system alike), and 0 means that the variables are not correlated. Five test subjects (two master students and three doctoral students, who were not part of the system development team) were involved in the experiment. Manual ambiguity ratings (integers ∈ [0, 4], i.e., ∈ [*min*, *max*] ambiguity) where acquired for 12-to-13 randomly pre-selected nodes per document, i.e., a total of 1000 nodes (during an average 10 hours rating time per tester) and then correlated with system ratings, computed with variations of *Amb_Deg*'s parameters to stress the impact of its factors ($Amb_{Polysemy}$, Amb_{Depth}, and $Amb_{Density}$): i) Test #1 considers all three factors equally ($w_{Polysemy} = w_{Depth} = w_{Density} = 1$), ii) Test #2 focuses on the polysemy factor ($w_{Polysemy} = 1$ while $w_{Depth} = w_{Density} = 0$), iii) Test #3 focuses on the depth factor ($w_{Depth} = 1$ while $w_{Polysemy} = 0.2$ and $w_{Density} = 0$), iv) Test #4 focuses on the density factor ($w_{Density} = 1$, $w_{Polysemy} = 0.2$ and $w_{Depth} = 0$).

Results compiled in Table 2 highlight several observations. First, XML ambiguity seems to be perceived and evaluated similarly by human users and our system – obtaining maximum positive correlation between human and *Amb_Deg* scores – when highly ambiguous and highly structured XML nodes are involved (e.g., Group 1). Second, ambiguity seems to be evaluated differently by users and our system when less ambiguous and/or poorly structured XML nodes are involved, attaining: negative or close to null correlation when low ambiguity and/or poorly structured XML nodes are evaluated (e.g., Groups 2, 3, and 4). This might be due to the intuitive understanding of semantic meaning by humans, in comparison with the intricate processing done by our automated system. For instance, in the case of documents of Dataset 9 of Group 4 (conforming to the *personnel.dtd* grammar of the Niagara XML document collection, cf. [38]), the meaning of child node label "*state*" under node label "*address*" was obvious for our human testers (providing an ambiguity score of 0/4). Yet, the interpretation of the meaning of "*state*" is not so obvious for a machine, especially using a rich lexical dictionary such as WordNet where word "*state*" has 8 different meanings. Here, a label considered relatively unambiguous form the user's point of view was assigned a higher ambiguity score by the system based on the expressiveness of the lexical reference.

Concerning *Amb_Deg*'s parameter weight variations (for $w_{Polysemy}$, w_{Depth}, and $w_{Density}$) with tests 2, 3, and 4, all three parameters seem to have comparable impacts on ambiguity evaluation. Note that evaluating XML node ambiguity is not addressed in existing approaches (they do not select target nodes, but simply disambiguate all of them, which can be complex and needless).

4.3 XML Semantic Disambiguation Quality

In addition to evaluating our XML ambiguity degree measure, we ran a series of experiments to evaluate the effectiveness of our XML disambiguation approach. We used the same test datasets described previously. Target XML nodes were first subject to manual disambiguation (12-to-13 nodes were randomly pre-selected per document yielding a total of 1000 target nodes, allowing the same human testers to manually annotate each node by choosing appropriate senses from WordNet, which required an average 22 hours per tester) followed by automatic disambiguation. We then compared user and system generated senses to compute *precision*, *recall* and *f-value* scores.

4.3.1 Testing with Different Configurations

We first tested the effectiveness of our approach considering its different features and possible configurations, considering: i) the properties of XML data (w.r.t. ambiguity and structure), ii) context size (sphere neighborhood radius), and iii) the disambiguation process used (concept-based, context-based, and the combined approach). We only show *f-value* levels in Figure 8 for space limitation (*precision* and *recall* levels follow similar patterns). Several interesting observations can be made here.

Table 2. Correlation between human ratings and system generated ambiguity degrees (cf. graphs in [38]).

		Test #1 *All factors*	**Test #2** *Polysemy*	**Test #3** *Depth*	**Test #4** *Density*
Group 1	Doc 1	0.394	0.411	0.335	**0.439**
Group 2	Doc 2	**0.017**	0.181	0.243	0.139
Group 3	Doc 3	-0.087	-0.139	-0.071	-0.138
	Doc 4	0.408	0.438	0.390	0.398
	Doc 5	-0.184	-0.185	-0.131	-0.235
Group 4	Doc 6	-0.284	-0.291	-0.243	-0.316
	Doc 7	-0.177	-0.190	-0.254	-0.143
	Doc 8	-0.119	-0.025	0.033	-0.156
	Doc 9	-0.452	-0.301	-0.251	**-0.456**
	Doc 10	-0.258	0.180	0.412	0.276

1) Considering ***XML data properties***, one can realize that our approach produced consistent *f-value* levels ∈ [0.55, 0.69] over all the tested configurations. The highest levels were reached with Dataset 1 of Goup1 having *high ambiguity* and *rich structure*, which resonates with the node ambiguity results discussed in the previous section (highly ambiguous and structurally rich XML nodes seem to be most effectively processed by our approach).

2) Considering ***context size***, optimal *f-value* levels are obtained with the smallest sphere neighborhood radius *d*=1 with Group 1 (*high ambiguity* and *rich structure* XML nodes), whereas optimal levels are obtained with larger contexts having *d*=3 with Groups 2, 3, and 4 (*low ambiguity* and/or *poor structure*). This is expected since increasing context size with highly ambiguous/structure rich XML would increase the chances of including noise (e.g., unrelated/heterogeneous XML nodes) in the disambiguation context and thus disrupt the process. Yet, increasing context size with less ambiguous/poorly structured XML could actually help in creating a large-enough and/or rich-enough context to perform effective disambiguation.

3) Considering the ***disambiguation process***, one can realize that the *concept-based* approach[12] generally produces higher *f-value* levels in comparison with the *context-based* approach, the latter appearing to be more sensitive to context size. This is expected since the context-based approach primarily depends on the notion of context and context nodes, in both the XML document and semantic network, and thus increasing/decreasing context size would disturb its effectiveness. The effect of context size here could be aggravated when using a rich semantic network (such as WordNet) where a small increase in sphere neighborhood radius could include a huge number of concepts (synsets) in the semantic network context vector, thus adding considerable noise.

To sum up, the above results emphasize the usefulness and need for a flexible approach (such as ours), allowing the user to fine-tune the disambiguation process in order to optimize disambiguation following the nature and properties of the data.

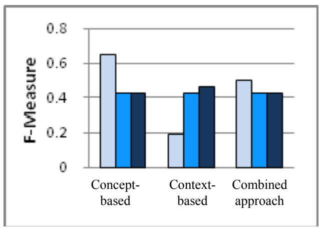
[12] When applying the concept-based approach, semantic similarity measures were considered with identical parameter weights ($w_{Edge} = w_{Node} = w_{Gloss} = 1/3 = 0.3334$), since evaluating the effectiveness of different semantic similarity measures is out of the scope of this paper.

Table 3. Characteristics of test documents.

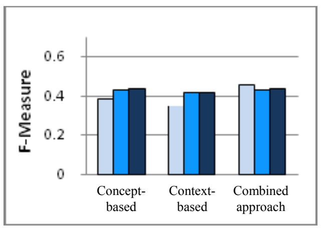
Groups	Datasets	Source dataset	Grammar	N# of docs	Avg. N# of nodes per doc	Node label polysemy (N# of senses)		Node Depth		Node Fan-out		Node Density	
						Avg.	Max.	Avg.	Max.	Avg.	Max.	Avg.	Max.
Group 1	1	Shakespeare collection[1]	shakespeare.dtd	10	192.054	7.052	30	3.687	6	0.604	20	0.38	6
Group 2	2	Amazon product files[2]	amazon_product.dtd	10	113.333	8.407	72	4.309	7	0.539	13	0.38	6
Group 3	3	SIGMOD Record[3]	ProceedingsPage.dtd	6	39.375	4.615	16	2.743	6	0.692	9	0.692	9
	4	IMDB database[4]	movies.dtd	6	15.475	4	10	2.666	5	1.066	5	1	5
	5	Niagara collection[5]	bib.dtd	8	26.5	4.384	13	2.961	5	0.884	5	0.884	5
Group 4	6	W3Schools[6]	cd_catalog.dtd	4	16.5	3.937	10	2.312	3	0.812	6	0.812	6
	7	W3Schools	food_menu.dtd	4	16	2.375	7	2.437	3	0.562	4	0.562	4
	8	W3Schools	plant_catalog.dtd	4	11.675	3.454	15	2	3	1.181	6	1.181	6
	9	Niagara collection	personnel.dtd	4	19	3.947	9	2.368	5	1.157	4	1.157	4
	10	Niagara collection	club.dtd	4	15.5	4.533	10	2.266	4	1.4	5	1.4	5

Table 4. Comparing our method with existing approaches

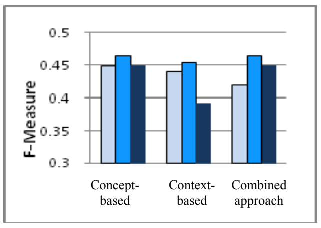
Approaches	Considers linguistic pre-processing	Considers tag tokenization (compound terms)	Addresses XML node ambiguity	Integrates an inclusive XML structure context	Flexible w.r.t. context size	Adopts *relational information* approach	Combines the results of various semantic similarity measures	Straightforward mathematical functions	Disambiguates XML structure and content
RPD [50]	√	x	x	x	x	x	x	x	x
VSD [29]	√	√	x	√	√	√	x	x	x
XSDF (our approach)	**√**	**√**	**√**	**√**	**√**	**√**	**√**	**√**	**√**



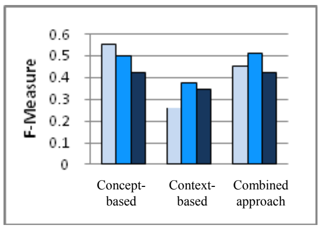
a. F-value results with Group 1.



b. F-value results with Group 2.



c. F-value results with Group 3.



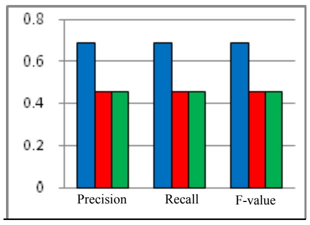
d. F-value results with Group 4.

▢ d = 1 🟦 d = 3 ■ d = 5 (or d=4 if *Max*(Depth(T)) = 4)

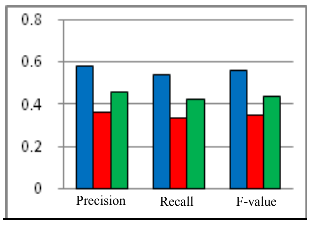
Figure 8. **Average *f-value* scores considering different features and configurations of our approach.**

4.3.2 *Comparative Study*

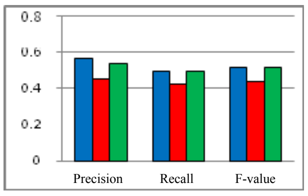
In addition, we evaluated the effectiveness of our approach in comparison with two of its most recent alternatives: *RPD* (Root Path Disambiguation) [50], and *VSD* (Versatile Structure Disambiguation) [29]. A qualitative comparison is shown in Table 4. We ran a battery of tests considering the different features and configurations or our approach. Here, we provide a compiled presentation considering optimal input parameters for our approach[19] (i.e., context size *d*=1 when processing Group 1, *d*=3 when processing Groups 2, 3, 4, using the *concept-based* disambiguation process with all groups) and its alternatives (as indicated in corresponding studies). Results in Figure 9 show that our method yields *precision, recall,* and *f-value* levels higher than those achieved by its predecessors, with almost all test groups except with Group 4 where *RPD* produces better results. In fact, XML nodes in Group 4 are less ambiguous and poorly structured in comparison with remaining test groups. Hence, choosing a simple context made of root path nodes has proven to be less noisy in this case, in comparison with the more comprehensive context models used with our approach and with VSD.



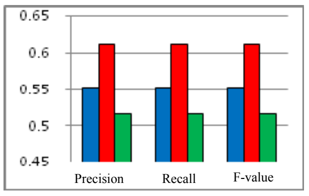
a. Results with Group 1.



b. Results with docs of Group 2.



c. Results with Group 3.



d. Results with Group 4.

🟦 XSDF (our approach) 🟥 RPD 🟩 VSD

Figure 9. **Average *PR*, *R* and *F-value* scores comparing our approach with RPD [50] and VSD [29].**

One can also realize that our method produces highest *precision*, *recall*, and *f-value* levels with Group 1 (*high ambiguity* and *rich structure* XML nodes), with an average 35% improvement over *RDP* and *VSD* (Figure 9.a), in comparison with average 25%, 5%, and almost 0% improvements with Groups 2, 3, & 4 respectively. This concurs with our results of the previous section: our method is more effective when dealing with highly ambiguous nodes within a rich XML structure, in comparison with less ambiguous/poorly structured XML.

[13] Available at http://metalab.unc.edu/bosak/xml/eg/shaks200.zip
[14] Available at simply-amazon.com/content/XML.html
[15] Available at http://www.acm.org/sigmod/xml
[16] Data extracted from http://www.imdb.com/ using a wrapper generator.
[17] Available at http://www.cs.wisc.edu/niagara/
[18] Available from http://www.w3schools.com
[19] Manually identified from repeated tests with different parameter values.

5. CONCLUSION

This paper introduces a novel XML Semantic Disambiguation Framework titled *XSDF*, to semantically annotate XML documents with the help of machine-readable lexical knowledge base (e.g., WordNet), which is a central pre-requisite to various applications ranging over semantic-aware query rewriting [11, 40], XML document classification and clustering [49, 53], XML schema matching [13, 55], and blog analysis and event detection in social networks [2, 7]. *XSDF* covers the whole disambiguation pipeline from: i) linguistic pre-processing of XML node labels to handle compound words (neglected in most existing solutions), to ii) selecting ambiguous nodes for disambiguation using a dedicated *ambiguity degree* measure (unaddressed in most solutions), iii) representing target node contexts as comprehensive and flexible (user chosen) *sphere neighborhood* vectors (in contrast with partial and fixed context representations, e.g., parent node or sub-tree context), and iv) running a hybrid disambiguation process, combining two (user chosen) methods: *concept-based* and *context-based* (in contrast with static methods). Experimental results w.r.t. user judgments reflect our approach's effectiveness in selecting ambiguous XML nodes and identifying node label senses, in comparison with existing solutions.

We are currently investigating different XML tree node distance functions (including edge weights, density, direction, etc. [16, 21]), to define more sophisticated neighborhood contexts. Fine-tuning user parameters using dedicated optimization techniques [19, 30] is another work in progress. We are also investigating the use of additional/alternative lexical knowledge sources such as Google [22], Wikipedia [12], and FOAF [2] to acquire a wider word sense coverage, and thus explore our approach in practical applications, namely semantic blog and wiki document clustering.

6. REFERENCES

[1] Abiteboul S. *et al.*, *Data on the Web: From Relations to Semistructured Data and XML.* Morgan Kaufmann Publisher, 1999. pp. 258.

[2] Aleman-Meza B. *et al.*, *Scalable Semantic Analytics on Social Networks for Addressing the Problem of Conflict of Interest Detection.* TWeb, 2008. 2(1):7.

[3] Amitay E. *et al.*, *Multi-Resolution Disambiguation of Term Occurrences.* Proc. of the Inter. CIKM Conf., 2003. pp. 255-262.

[4] Artiles J. *et al.*, *Word Sense Disambiguation based on Term to Term Similarity in a Context Space.* In SensEval-3:, 2004. pp. 58-63.

[5] Banerjee S. and Pedersen T., *An adapted Lesk algorithm for word sense disambiguation using WordNet.* In Proc. of CICLing'02, 2002.

[6] Banerjee S. and Pedersen T., *Extended Gloss Overlaps as a Measure of Semantic Relatedness.* Inter. IJCAI'03 Conf., 2003. p. 805-810.

[7] Berendt B. *et al.*, *Bridging the Gap: Data Mining and Social Network Analysis for Integrating Semantic Web and Web 2.0.* JWS, 2010. 8(2-3): 95-96.

[8] Bertino E. *et al.*, *Measuring the structural similarity among XML documents and DTDs.* J. of Intel. Info. Systems, 2008. 30(1):55-92.

[9] Budanitsky A. and Hirst G., *Evaluating WordNet-based Measures of Lexical Semantic Relatedness.* Compt. Linguistics, 2006. 32(1): 13-47.

[10] Chan Y. S. *et al.*, *NUS-PT: Exploiting parallel texts for word sense disambiguation in the English all-words tasks.* SemEval, 2007, 253-256

[11] Cimiano P. *et al.*, *Towards the Self-Annotating Web.* WWW, 2004, 462-471.

[12] Dandala B. *et al.*, *Sense Clustering using Wikipedia.* RANLP, 2013, 164-171.

[13] Do H. and Rahm E., *Matching Large Schemas: Approaches and Evaluation.* Information Systems, 2007. 32(6): 857-885.

[14] Fellbaum C., *Wordnet: An Electronic Lexical Database.* MIT Press, 1998, 422.

[15] Francis W. N. and Kucera H., *Frequency Analysis of English Usage.* Houghton Mifflin, Boston, 1982.

[16] Ganesan P. *et al.*, *Exploiting Hierarchical Domain Structure To Compute Similarity.* ACM TOIS, 2003. 21(1):64-93.

[17] Graupmann J. *et al.*, *The SphereSearch Engine for Unified Ranked Retrieval of Heterogeneous XML and Web Documents.* VLDB, 2005, 529-540.

[18] Guo Y. *et al.*, *HIT-IR-WSD: A WSD System for English Lexical Sample Task.* SemEval 2007, ACL.

[19] Hopfield J. and Tank D., *Neural Computation of Decisions in Optimization Problems.* . Biological Cybernetics, 1985. 52(3):52–141.

[20] Ide N. and Veronis J., *Introduction to the Special Issue on Word Sense Disambiguation: The State of the Art.* Compt. Ling., 1998. 24(1):1-40.

[21] Jiang J. and Conrath D., *Semantic Similarity based on Corpus Statistics and Lexical Taxonomy.* Inter. COLING Conf., 1997.

[22] Klapaftis I. and Manandhar S., *Evaluating Word Sense Induction and Disamiguation Methods.* Language Resources and Eval., 2013.

[23] Krovetz R. and Croft W. B., *Lexical Ambiguity and Information Retrieval.* ACM Trans. on Info. Systems, 1992. 10(2):115-141.

[24] Leacock C. and Chodorow M., *Combining Local Context and WordNet Similarity for Word Sense Identification.* MIT Press, 1998. pp. 265-283.

[25] Lee J. *et al.*, *Information Retrieval Based on Conceptual Distance in IS-A Hierarchies.* J. of Documentation, 1993. 49(2):188-207.

[26] Lesk M., *Automatic Sense Disambiguation using Machine Readable Dictionaries.* Inter. SIGDOC'86 Conf., 1986.

[27] Lin D., *An Information-Theoretic Definition of Similarity.* ICML, 1998.

[28] Maguitman A. *et al.*, *Algorithmic Detection of Semantic Similarity.* Proc. of the Inter. WWW Conf., 2005. pp. 107-116.

[29] Mandreoli F. *et al.*, *Versatile Structural Disambiguation for Semantic-Aware Applications.* In Proc. of Inter. CIKM Conf., 2005. pp. 209-216.

[30] Marie A. and Gal A., *Boosting Schema Matchers.* OTM 2008, 283-300.

[31] Marquez L. *et al.*., *Supervised corpus-based methods for WSD. In Word Sense Disambiguation: Algorithms and Applications.* E. Agirre and P. Edmonds, Eds. Springer, New York, NY, 2006. pp. 167-216.

[32] McGill M., *Introduction to Modern IR,* 1983. McGraw-Hill, New York.

[33] Mihalcea R., *Knowledge-based Methods for WSD.* In Word Sense Disambiguation: Algorithms and Applications, 2006. pp. 107–131.

[34] Mihalcea R. and Faruque E., *Senselearner: Minimally supervised word sense disambiguation for all words in open text.* SensEval-3, 2004, 155-158.

[35] Miller G., *WordNet: An On-Line Lexical Database.* Inter. Journal of Lexicography, 1990. 3(4).

[36] Miller G.A. *et al.*, *A Semantic Concordance.* In Proc. of ARPA Workshop on Human Language Technology, 1993. pp. 303–308.

[37] Ming M. *et al.*, *A Harmony Based Adaptive Ontology Mapping Approach.* In Proc. of the Inter. SWWS'08 Conf., 2008. pp. 336-342.

[38] Charbel N. *et al.*, *Resolving XML Semantic Ambiguity - Technical Report.* Available at http://sigappfr.acm.org/Projects/XSDF/, 2014.

[39] Navigli R., *Word Sense Disambiguation: a Survey.* CSUR, 2009. 41(2):1–69.

[40] Navigli R. and Velardi P., *An Analysis of Ontology-based Query Expansion Strategies.* In proc. of the Inter. IJCAI'03 Conf., 2003.

[41] Nierman A. and Jagadish H. V., *Evaluating structural similarity in XML documents.* Proc. of WebDB, 2002. pp. 61-66.

[42] Patwardhan S. *et al.*, *Using Measures of Semantic Relatedness for Word Sense Disambiguation.* In Proc. of CICLing'03, 2003. pp. 241-257.

[43] Patwardhan S. *et al.*, *SenseRelate:TargetWord – A Generalized Framework forWord Sense Disambiguation.* AAAI'05, 4, 1692-1693.

[44] Pradhan S. *et al.*, *Semeval-2007 task-17: English lexical sample, SRL and all words.* In Proc. of SemEval'07, 2007. pp. 87-92.

[45] Resnik P., *Disambiguating Noun Groupings with Respect to WordNet Senses.* 3rd Workshop on Large Corpora, 1995. pp. 54-68.

[46] Richardson R. and Smeaton A., *Using WordNet in a Knowledge-based approach to information retrieval.* BCS-IRSG Colloquium on IR, 1995.

[47] Stanford Center for Biomedical Informatics Research. *Protégé Ontology Editor.* [cited January 2015].

[48] Tagarelli A. *et al.*, *Word Sense Disambiguation for XML Structure Feature Generation.* In Proc. of the ESWC, 2009, pp. 143–157.

[49] Tagarelli A. and Greco S., *Semantic Clustering of XML Documents.* ACM Transactions on Information Systems (TOIS), 2010. 28(1):3.

[50] Tagarelli A. *et al.*, *Word Sense Disambiguation for XML Structure Feature Generation.* In Proc. of ESWC, 2009, pp. 143–157.

[51] Taha K. and Elmasri R., *CXLEngine: A Comprehensive XML Loosely Structured Search Engine.* Proc. of the EDBT DataX'08, 2008, 37-42.

[52] Taha K. and Elmasri R., *XCDSearch: An XML Context-Driven Search Engine.* IEEE TKDE, 2010. 22(12):1781-1796.

[53] Tekli J. *et al.*, *A Novel XML Structure Comparison Framework based on Sub-tree Commonalities and Label Semantics.* Elsevier JWS, 2012. 11: 14-40.

[54] Tekli J. *et al.*, *An Overview of XML Similarity: Background, Current Trends and Future Directions.* Elsevier CS Review, 2009. 3(3):151-173.

[55] Tekli J. *et al.*, *Minimizing User Effort in XML Grammar Matching.* Elsevier Information Sciences Journal, 2012. 210:1-40.

[56] Theobald M. *et al.*, *Exploiting Structure, Annotation, and Ontological Knowledge for Automatic Classification of XML Data.* WebDB, 2003, pp. 1-6.

[57] Tratz S. *et al.*, *PNNL: A supervised maximum entropy approach to word sense disambiguation.* In Proc. of SemEval, 2007. pp. 264-267.

[58] W3C. *The Document Object Model*, 2005, http://www.w3.org/DOM.

[59] Wu Z. and Palmer M., *Verb Semantics and Lexical Selection.* 32nd Annual Meeting of the ACL, 1994. pp. 133-138.

[60] Yaworsky D., *Word-Sense Disambiguation Using Statistical Models of Roget's Categories Trained on Large Corpora.* COLING, 1992, pp. 454-460.

[61] Zhang Z. *et al.*, *Similarity Metric in XML Documents.* Knowledge Management and Experience Management Workshop, 2003.

SpMachO - Optimizing Sparse Linear Algebra Expressions with Probabilistic Density Estimation

David Kernert
Technische Universität Dresden
Database Technology Group
Dresden, Germany
david.kernert@sap.com

Frank Köhler
SAP SE
Dietmer-Hopp-Allee 16
Walldorf, Germany
frank.koehler@sap.com

Wolfgang Lehner
Technische Universität Dresden
Database Technology Group
Dresden, Germany
wolfgang.lehner@tu-dresden.de

ABSTRACT

In the age of statistical and scientific databases, there is an emerging trend of integrating analytical algorithms into database systems. Many of these algorithms are based on linear algebra with large, sparse matrices. However, linear algebra expressions often contain multiplications of more then two matrices. The execution of sparse matrix chains is nontrivial, since the runtime depends on the parenthesization and on physical properties of intermediate results. Our approach targets to overcome the burden for data scientists of selecting appropriate algorithms, matrix storage representations, and execution paths. In this paper, we present a sparse matrix chain optimizer (SpMachO) that creates an execution plan, which is composed of multiplication operators and transformations between sparse and dense matrix storage representations. We introduce a comprehensive cost model for sparse-, dense- and hybrid multiplication kernels. Moreover, we propose a sparse matrix product density estimator (SpProdest) for intermediate result matrices. We evaluated SpMachO and SpProdest using real-world matrices and random matrix chains.

Categories and Subject Descriptors

G.1.3 [**Numerical Linear Algebra**]: Sparse, structured, and very large systems

General Terms

sparse linear algebra, optimization

1. INTRODUCTION

In the era of big data and data deluge, scientists and data analysists are confronted with a time-consuming implementation overhead, when they want to scale and speed up their existing, handcrafted code that has been working for years on small data sets. This set the way for new database applications in the fields of scientific computations and advanced analytics on large data. However, since most of the algorithms in science and data mining are composed of linear algebra expressions, conventional SQL-based relational database management systems (RDBMS's) did not match the requirements. On the other hand, numerical algebra systems like R are known for efficient linear algebra algorithms, but they lack scalability and data manipulation capabilities. As a consequence, the demand of data scientists for a scalable system that provides a basic set of efficient linear algebra primitives attracted the attention of the database community [11]. Recently emerged systems like SciDB [8] or SystemML [6] reacted by providing a R or R-like interface and deep integrations of linear algebra primitives, such as sparse matrix-matrix and matrix-vector multiplications.

In business environments, data analysts often load data from a relational database into a numerical algebra system to perform their analysis by means of linear algebra. For example, financial analysts that use a RDBMS for storing stock price data need the functionality of SQL, e.g., in order to get the average stock prizes, or to find all stocks that belong to a certain group. On the other side, they might want to use matrix multiplications to find correlations [24] of stocks and derivatives.

Matrices as first-class citizens have been integrated in many systems, for example in array DBMS's [8], data warehouses [19], or in-memory column stores [20], but only little has been done in the direction of optimizing the execution of linear algebra expressions. In this paper, we focus on optimizing the execution of sparse linear algebra expressions based on physical properties. The idea is the following: next to the RDBMS optimizer that creates optimized execution plans from SQL expressions, we propose a component (Fig. 1) that generates an optimal execution plan for linear algebra, which is using the native storage and execution engine of the system, and additional operators for linear algebra, such as a matrix multiplication operator.

As most of the big matrices occurring in the real world are *sparse*, linear algebra expressions often contain multiplications of three or more sparse, or mixed dense and sparse matrices, e.g. in transitive closure computations, Markov chains [27], linear transformation [13], or linear discrete dynamical systems [4]. An efficient execution of sparse matrix chain multiplications is nontrivial, in particular, if intermediate result matrices become dense. In many situations

10.5441/002/edbt.2015.26

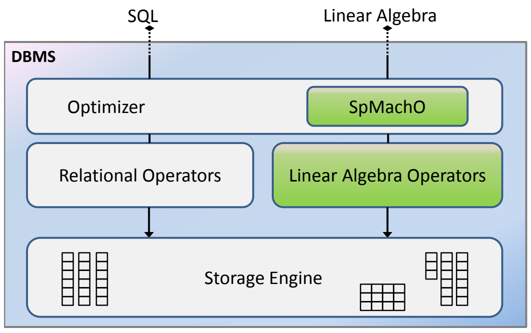


Figure 1: DBMS example architecture.

the runtime performance can be significantly improved by changing the execution order, or by switching from a sparse to a dense multiplication algorithm in later stages. Since data scientists are usually not familiar with algorithmic details of multiplication kernels and system parameters, and do not have profound knowledge of the characteristics of their matrices, it makes sense to leave these decisions to the system.

In particular, the contributions of this work are:

- **SpMachO**, a general matrix chain multiplication optimizer based on a dynamic programming approach, which leverages density estimations of intermediate results and different multiplication kernels to minimize the total execution runtime. The optimization problem and the SpMachO algorithm are presented in sections 2 and 3.
- **SpProdest**, a sparse matrix density estimator, which predicts the density structure of intermediate and final result matrices, by using a novel skew-aware stochastic density propagation method. It is described in detail in sections 4 and 5.
- An extensive evaluation and comparison of the execution runtime of the SpMachO-generated plan against alternative execution strategies and other numerical algebra systems, which is presented in section 6.

Finally, we will discuss the related work in section 7, followed by the conclusion in section 8.

2. EXPRESSION OPTIMIZATION

A lot of data scientists work with numeric algebra systems to run their linear algebra algorithms. However, the user is often let alone with the execution order, although the way of executing large sparse matrix expressions contains a significant optimization potential.

Consider a set of linear algebra expressions that consist of matrix multiplication and addition on general $\mathcal{R}^{m\times n}$ matrices. Further operations, like subtraction or division by a matrix $\boldsymbol{A}$ can be represented by using the corresponding inverse of addition $(-\boldsymbol{A})$, or inverse of multiplication $\boldsymbol{A}^{-1}$, respectively. Thus, the expression can be reduced to a form:

$$\boldsymbol{C} = \boldsymbol{A}_0 + \boldsymbol{A}_1 \cdot \boldsymbol{A}_2 \cdot ... \cdot \boldsymbol{A}_p + \boldsymbol{A}_{p+1} \cdot ...\boldsymbol{A}_l + ...,$$

From a mathematical perspective, the number of operations needed for the element-wise addition of intermediate results, or any element-wise operation in general, is independent from the execution order, thus, it is the same for $(\boldsymbol{A}+\boldsymbol{B})+\boldsymbol{C}$ as

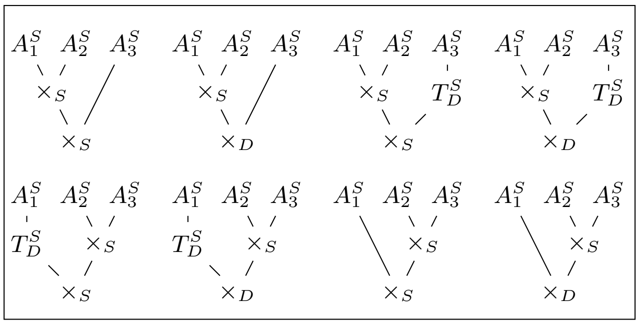


Figure 2: Eight of the possible 128 execution plans for the multiplication of three sparse matrices $\boldsymbol{A}_1 \cdot \boldsymbol{A}_2 \cdot \boldsymbol{A}_3$. $\times$ are binary multiplication operators, S/D denotes the internal *sparse/dense* representation type of matrices, and T are unary storage type transformations of matrices.

for $\boldsymbol{A}+(\boldsymbol{B}+\boldsymbol{C})$. Although this indifference might not hold in practise, when different physical representations are used, the addition part in the computation of $\boldsymbol{C}$ is rather cheap, since the complexity is at most $\mathcal{O}(mn)$ for dense matrices. Most of the execution time is spent in the computation of multiplications

$$\boldsymbol{A}_1 \cdot \boldsymbol{A}_2 \cdot ... \cdot \boldsymbol{A}_{p-1} \cdot \boldsymbol{A}_p, \quad \boldsymbol{A}_i \in \mathcal{R}^{m_i \times m_{i+1}}, \tag{1}$$

so the work of this paper focus on the optimization of matrix chain multiplications. The algebraic degree of freedom to execute expression (1) consists in the setting of parenthesis, since matrix multiplications are associative. Altering the parenthesization does not change the result, but the number of operations required in the computation of the complete chain can vary significantly. Finding the optimal parenthesization for a dense, non-square matrix chain multiplication is well understood and serves as a text book example for the use of dynamic programming [16, 12]. The idea is to iteratively construct the optimal parenthesization as a combination of optimal sub-parenthesizations, by minimizing a cost recurrence expression with respect to the split point k

$$C_{\pi^{\mathrm{B}}(ij)} = \min_{i\leq k<j} \{C_{\pi^{\mathrm{B}}(ik)} + C_{\pi^{\mathrm{B}}((k+1)j)} + \mathtt{TM}\left(\boldsymbol{A}_{[i...k]}, \boldsymbol{A}_{[k+1...j]}\right)\}. \tag{2}$$

For the mathematical formulation, we introduce

- $\pi(ij)$: a parenthesization for the matrix (sub)chain $\boldsymbol{A}_{[i...j]}$
  $\pi^{\mathrm{B}}(ij)$ denotes the optimal ("best") one.
- C_π: the cost for executing the matrix chain multiplication, given a certain parenthesization π.
- $\mathtt{TM}\left(\boldsymbol{A}_{[i...k]}, \boldsymbol{A}_{[k+1...j]}\right)$: the cost function for multiplying the two matrices that result from the subchains $\boldsymbol{A}_{[i...k]}$ and $\boldsymbol{A}_{[k+1...j]}$

In the textbook case, the combination cost $\mathtt{TM}\left(\boldsymbol{A}_{[i...k]}, \boldsymbol{A}_{[k+1...j]}\right)$ is set equal with the number of flops for multiplying the two dense intermediate result matrices. By using the classical inner product algorithm the costs can be exactly determined a priori as $m_i m_{k+1} m_{j+1}$.

However, the dense matrix case has certain limitations that restricts its relevance in practice. Most important, many of the real-world matrices in big data environments are sparse. A sparse matrix is not only defined by its row and column dimensions m and n, but also by the number and the pattern of non-zero elements N_{nz}, or the *density*[1] $\rho = \frac{N_{nz}}{mn}$. As a matter of fact, algorithms on sparse matrices have a different complexity: unlike the naive inner product algorithm for dense matrices, the cost of a multiplication of two sparse matrices rather depends on the number of non-zero elements than on their dimensions. Moreover, sparse matrices are stored in a different data structure than dense matrices, which leads to different actual costs depending on the characteristics of the physical representation. Most of the related work on matrix chain multiplications consider either dense-only [16, 12] or sparse-only [9] multiplications, being agnostic to the fact that the densities of the intermediate result matrices can vary significantly from the initial matrices. For example, the density of the result matrix $\boldsymbol{C} = \boldsymbol{A} \cdot \boldsymbol{B}$ can be much higher, or even less than that of both $\boldsymbol{A}$ and $\boldsymbol{B}$ (see Fig. 5). Despite the mathematical complexity, it is in many cases more efficient to continue using algorithms on dense matrix representations, if the density exceeds a certain threshold. This can be reasoned with the efficient and well-tuned implementations of dense matrix multiplication kernels in BLAS[2].

Our idea is to take the individual differences of the different matrix representations and multiplication kernels into account, and to exploit the potential performance benefits from changing the physical implementation of the initial matrices or intermediate results. We construct an execution plan for the chain expression that can contain dense, sparse and mixed dense/sparse matrix multiplications. Furthermore, the execution plan can contain conversions from a sparse into a dense representation. Therefore, we adopted the idea of dynamic programming and modified it in such a way that it incorporates the physical properties of the matrices. We extended the recurrence (2) by adding the input and output storage types as independent dimensions, and added cost functions for the storage type conversions:

$$\begin{aligned} &C_{\Pi^{\mathrm{B}}(ij)S^o} = \\ &\min_{\substack{i \le k < j \\ S^l, S^r, S^1, S^2 \in \mathcal{S}}} \Big\{ C_{\Pi^{\mathrm{B}}(ik)S^l} + C_{\Pi^{\mathrm{B}}((k+1)j)S^r} + \mathtt{TT}_{S^1}\left(\boldsymbol{A}_{[i..k],S^l,\rho}\right) \\ &+ \mathtt{TT}_{S^2}\left(\boldsymbol{A}_{[k+1..j],S^r,\rho}\right) + \mathtt{TM}_{S^o}\left(\boldsymbol{A}_{[i..k],S^1,\rho}, \boldsymbol{A}_{[k+1..j],S^2,\rho}\right) \Big\} \end{aligned} \tag{3}$$

- $\Pi(ij)$: execution plan for a matrix (sub)chain multiplication. It contains the execution order as well as all storage transformations. Π^{B} denotes the optimal plan.

- S^X: storage type, which is either dense or sparse. The superscript X labels each of the five matrices that are considered per execution node. $X = l$: left subplan output-, r: right subplan output-, 1: left input-, 2: right input-, o: current product output matrix .

- $\mathtt{TT}_{S^Y}\left(\boldsymbol{A}_{[i..k],S^X}\right)$: cost function for the conversion of a matrix from type S^X into type S^Y.

[1]In the remainder of this paper, we refer to the non-zero structure using ρ rather than N_{nz}

[2]Basic Linear Algebra Subprograms, http://www.netlib.org/blas/

Since the cost functions TM in equation (3) depend on the storage types of the input and output matrices, as well as their densities, it could be beneficial to convert a matrix from one into the other representation prior to the multiplication because conversions are usually less costly than multiplications. For example, if the initial matrices are in a sparse representation, and the dense multiplication kernel plus the conversion has a far lower cost than the sparse multiplication, then they are first converted into the dense representation. As a matter of fact, the value of the conversion cost $\mathtt{TT}_S(A, S)$ equals zero for identity transformations, i.e., when a matrix is already in the optimal representation. Hence, besides the parenthesization split point k, we vary the input and output storage types for each step in recurrence (3).

Some of the parameters that contribute to the cost functions $\mathtt{TM}/\mathtt{TT}(\cdot)$, for example the density ρ of intermediate results, are not known prior to the execution and have to be estimated. Therefore, we developed the sparse matrix product density estimator SpProdest, which is described in section 4 and 5 of this paper. The resulting costs derived from recurrence (3) are minimal, given that the estimated costs encoded in TM and TT are determined precisely. In particular, the optimality, or *goodness*, of SpMachO depends on two parts, which potentially contain uncertainties: first, the accuracy of the quantitative cost model of the multiplication kernels, and second, the precision of the density estimates provided by SpProdest.

The total number of the possible execution plans using our model (3) for a matrix chain multiplication of length p is

$$\mathcal{C}_{p-1} \cdot 2^{3(p-1)}, \tag{4}$$

where $\mathcal{C}_{p-1}$ denotes the Catalan number $\mathcal{C}_n = \frac{(2n)!}{(n+1)!n!}$.

$\mathcal{C}_{p-1}$ reflects the number of possible parenthesizations, which is the same as for the textbook case [12]. The second factor is related to the 2^3 {left input-, right input-, output-} storage type combinations that are connected with each of the $p-1$ type multiplication nodes. The number in (4) resembles the size of the search space, which grows exponentially. To give an example, it yields 2560 for a matrix chain of length $p = 4$ and already 1376256 for $p = 6$. As in [12], SpMachO solves the recurrence (3) in $\mathcal{O}(p^3)$ time using a bottom-up dynamic programming approach.

3. SpMachO

The pseudocode of SpMachO is sketched in Algorithm 1. The cost of the optimal sub-chain multiplications and the relevant plan information (split points and storage types per sub-chain) are cached in three-dimensional array structures. For each combination in the inner loop, the method chkMemLimit checks if the total memory consumption of the matrices and intermediate results in the current plan configuration would exceed the system limit. The memory required for dense $m \times n$ matrices is $\mathcal{O}(mn)$, and $\mathcal{O}(N_{nz} = mn\rho)$ for sparse matrices. Since SpMachO optimizes the runtime performance, the plan may contain conversions from sparse into dense matrix representations whenever the dense kernel leads to a lower overall runtime, due to the efficient dense kernel implementation. However, the conversions into dense representations potentially increase the memory consumption compared to a sparse-only plan. Our strategy is that every conversion is allowed, as long as the total memory consumption at every point in time does not exceed a hard memory

Algorithm 1 SpMachO

```
 1: function spMacho(MatrixChain $\boldsymbol{A}_{[1..p]}$, TM, TT)
 2:     $\hat{\boldsymbol{\rho}}[][] \leftarrow$ spProdest($\boldsymbol{A}_{[1..p]}$)
 3:     for $1 \leq j < p, j > i \geq 0$ do
 4:         for $i \leq k < j$ do
 5:             for types $\in$ {sparse, dense} do
 6:                 if !chkMemLimit($A, k, \rho$, types) then
 7:                     continue
 8:                 $q \leftarrow$ TT($\boldsymbol{A}_{[i..k..j]}, \hat{\boldsymbol{\rho}}[][]$, types)
 9:                 $q \leftarrow q +$ TM($\boldsymbol{A}_{[i..k..j]}, \hat{\boldsymbol{\rho}}[][]$, types)
10:                 if $q <$ cost[$i$][$j$][$S^o$] then
11:                     cost[$i$][$j$][$S^o$] $\leftarrow q$
12:                     plan[$i$][$j$][$S^o$] $\leftarrow$ k, types
13:     if cost[1][$p$][$\cdot$] $\geq$ MAXVAL] then
14:         /* memory exceed exception */
15:     else
16:         return min(plan[1][p][sparse], plan[1][p][dense])
```

limit. Execution paths that would exceed the memory limit are automatically skipped (line 6). We assume that there is at least one plan that does not exceed the memory limit. If not, SpMachO returns with an exception (line 13). Finally, the resulting plan, which can be converted into a directed acyclic graph representation as of Fig. 2, is returned to the system for execution.

The complexity of Algorithm 1 is $\mathcal{O}(p^3)$, which can be derived from the dynamic programming loop and is the same as in the dense-only problem. The additional complexity by the introduction of storage type transformations yields a constant factor, since the inner loops over the storage types do not depend on the chain length p. A few pruning methods can be applied to reduce the execution plan space, for example, by excluding that the product of two dense can be sparse. However, they do not lower the asymptotic complexity of the algorithm.

The system executes the plan using the corresponding transformation and matrix multiplication operators. While the unary transformation operator either performs a dense-to-sparse or a sparse-to-dense storage transformation, the eight-fold multiplication operator delegates the execution to one of the multiplication kernels. We implemented all algorithms in our prototype using row-major 2D-arrays for dense- and the columnar compressed sparse row layout (CSR) for sparse matrices, since they are to our notion the most common physical representations, and are used in many numerical libraries, e.g. the Intel Math Kernel Library [1]. As mentioned in the beginning, we already showed in [20] that these representations can be mapped onto a columnar storage layout of an in-memory columnar DBMS.

3.1 Multiplication Kernels

There are eight different **ge**neral **m**atrix **m**ultiply (gemm) kernels that are used in our system. We will use the notation *xyz_gemm* to denote a multiplication kernel, where x is the left-hand input-, y is the right-hand input type and z the output matrix storage type, which can be either sparse (*sp*) or dense (*d*). Some of the kernels, for example the standard BLAS *ddd_gemm*, or *spdd_gemm*, are implemented by vendor-tuned C++ libraries, so we call the library instead of providing an own implementation. As of the current status, this is done only for *ddd_gemm*, for which we use the Intel MKL implementation.

Table 1: The matrix multiplication kernels for the product $C^{m\times n} = A^{m\times k} \cdot B^{k\times n}$ and their cost functions used in SpMachO. $N_\times$ denotes the number of actual multiplications, the hat indicates the corresponding estimated value. α, β, γ are constant parameters.

Kernel	Cost Function
ddd_gemm	$\alpha(mkn)$
spdd_gemm	$\alpha\hat{N}_\times$
dspd_gemm	$\alpha(mk) + \beta\hat{N}_\times$
spspd_gemm	$\alpha N^A_{\text{nz}} + \beta\hat{N}_\times$
ddsp_gemm	$\alpha\hat{N}_\times + \beta\hat{N}^C_{\text{nz}} + \gamma(mn)$
spdsp_gemm	$\alpha N^A_{\text{nz}} + \beta\hat{N}_\times + \gamma\hat{N}^C_{\text{nz}}$
dspsp_gemm	$\alpha\hat{N}_\times + \beta\hat{N}^C_{\text{nz}} + \gamma(mk)$
spspsp_gemm	$\alpha N^A_{\text{nz}} + \beta\hat{N}_\times + \gamma\hat{N}^C_{\text{nz}}$

Table 1 lists the kernels that are used in our system. In order to obtain the optimal execution plan via solving recurrence (3), the cost model and its corresponding parameters have to be determined accurately for each multiplication kernel. Since the actual runtime depends on many parameters and external influences, an exact determination cannot be guaranteed. However, even for small variations, the plan generated by SpMachO is still near-optimal, which we verified in the evaluation in section 6.

3.2 Execution Time Cost Model

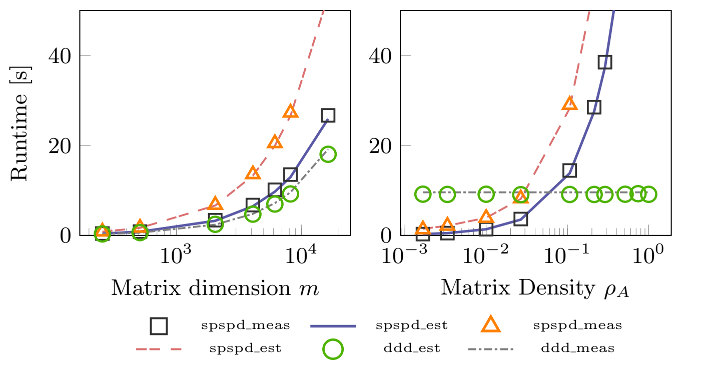
The cost for multiplying two dense or sparse matrices generally depends on the matrix dimensions m, k, n, the number and pattern of non-zero elements of both the input and the result matrices, and the implementation details of the corresponding kernel algorithm. The idea is to reduce the dimensions to a set of only a few, significant dimensions, and create the cost model based on the reduced dimension set. On average, it is a fair approximation to assume that the runtime of a single multiplication only depends on the number of non-zero elements, and not on the individual non-zero pattern variations. Hence, we are able to reduce the parameter space to the dimensions m, k, n, ρ_A, ρ_B, and $\hat{\rho}_C$, which corresponds to the product density estimation, which is determined by SpProdest.

As an example, we will examine the *spspsp_gemm* kernel that uses the popular Gustavson algorithm [18]. The algorithm is based on the *sparse accumulator* method, which is still commonly used for sparse matrix implementations [15]. In order to not repeat the algorithm description in detail, we only sketch the derivation of our cost model for *spspsp_gemm*, which we consider as the most interesting kernel.

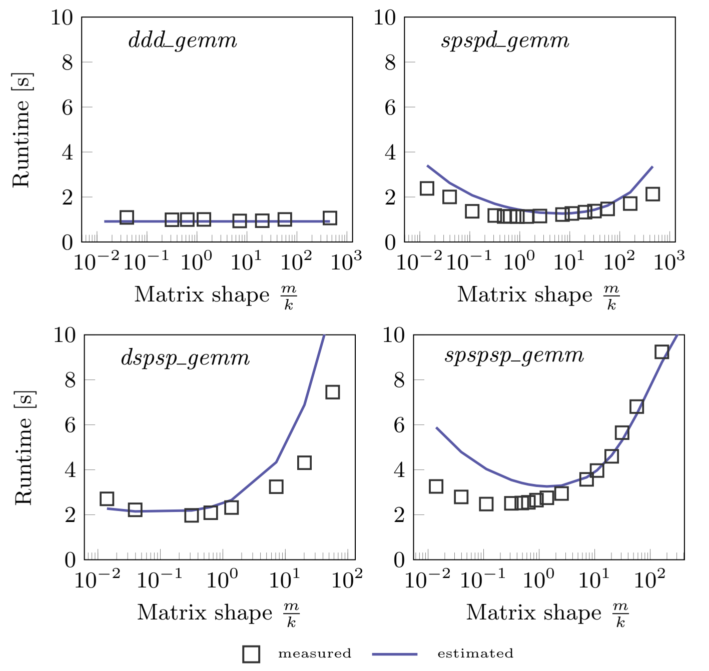
For reasonably large sparse matrices, the runtime of sparse kernels is often dominated by main memory bandwidth. Hence, mainly the memory accesses contribute to the runtime of an algorithm, which is in conformance to the external memory (I/O) model. For *spspsp_gemm*, the access pattern can be formulated as

$$\texttt{TM}(m, k, n, \rho_A, \rho_B) = m \times (\text{read}_{rowA} + k\rho_A \times (\text{read}_{colA,rowB} + n\rho_B \times (\text{read}_{colB,valA,valB} + \text{write}_{colC,valC})) + n\hat{\rho}_C \text{write}_{colC,valC}),$$

where the read/write denote the accesses to the respective data structures (*rowA, rowB,* etc.) in memory. For example,



(a) Scaling behavior: varying the dimension (left) and density (right). Other dimensions are fixed: $k = 8192, n = 8192, \rho_B = 0.1$



(b) Varying the matrix shape m/k. $N_\times$ is fixed in each plot.

Figure 3: Measured runtimes (markers) and time estimates (lines) for different matrix multiplication kernels.

a read on *colA* has a higher average cost than a read on *colB*, since a complete row of matrix $\boldsymbol{B}$ is touched in-between two consecutive *colA*-reads, thus, the system has most probably evicted the cache line of *colA* and has to fetch it again. The exact number of cycles, and hence, the required time per read or write access depends on whether the addressed cache line resides in the system cache. However, since we consider large matrices with sizes that are by factors larger than the last level system cache, we approximate the read and write accesses as fixed time constants in our model. Moreover, instead of determining the individual time constants for the $\text{read}_X/\text{write}_X$ access, we abstracted them into few parameters, which can then be determined empirically. Therefore, we expand the expression of `TM` and accumulate the time constants of the read/write accesses into the constant parameters α, β, γ, and obtain a simple time approximation for the

Wall-clock time

$$T \approx \alpha(\underbrace{m \cdot k \cdot \rho_A}_{N^A_{\text{nz}}}) + \beta(\underbrace{m \cdot k \cdot \rho_A \cdot n \cdot \rho_B}_{\hat{N}_\times}) + \gamma(\underbrace{m \cdot n \cdot \hat{\rho}_C}_{\hat{N}^C_{\text{nz}}}),$$

which only depends on the constant parameters

$$\alpha = T(\mathit{read}_{colA,rowB})$$
$$\beta = T(\mathit{read}_{colB,valA,valB} + \mathit{write}_{colC,valC})$$
$$\gamma = T(\mathit{write}_{colC,valC})$$

and the *derived* dimensions:

- N^A_{nz}: the number of non-zeros in matrix $\boldsymbol{A}$
- $\hat{N}_\times$: the estimated number of multiplications
- $\hat{N}^C_{\text{nz}}$: the estimated number of non-zero elements in the result matrix

For the other kernels, the cost function can be deduced in a similar manner. Because of space limitations, we will not discuss them in this paper. Table 1 lists the cost function for each kernel. Each cost function is a linear combination of different derived dimensions, weighted by constant parameters α, β, γ. The constant parameters are estimated for each kernel by a multilinear least-squares fit. Since they are dependent on the system hardware, the fit has to be done once for each system configuration.

Fig. 3a shows the scaling behavior of the matrix multiplication kernels *spspsp_gemm*, *spspd_gemm* and *ddd_gemm* with respect to matrix dimension and density. We observe that our cost models (lines) conform well with the actual algorithm runtimes (markers). From the right plot in Fig. 3a we can infer the following example scenarios: if the density ρ_A has a higher value than about 0.1, then it can be worthwhile to convert A into a dense representation and continue with the dense kernel – of course only if the dense representation does not exceed the available memory. If the density is below, it is probably best to select the *spspd_gemm* kernel. For $\rho_A \ll 0.01$ and depending on the following multiplications and the estimated intermediate result densities, it might be best to take with the *spspsp_gemm* kernel.

In Fig. 3b, we fixed the densities ρ_A, ρ_B and the product $(m \cdot k \cdot n)$, and only varied the relative matrix shapes $\frac{m}{k} \equiv \frac{k}{n}$. At both edges of each plot the matrices are extremely rectangular. The deviation of the actual times from our estimates shows that our cost model has limited accuracy in these extreme cases. However, the deviation is still acceptable, since the times are always overestimated. Overestimation is more robust than underestimation, because SpMachO would then just select another multiplication kernel, whereas in the latter case the deviation would propagate into the overall estimation. It is worthwhile mentioning that one conclusion we deduce from Fig. 3b is that simplistic cost models, which solely depend on the number of non-zero multiplications $N_\times$, are not able to describe the shape dependency, since $N_\times \equiv$ *const.* is fixed in each plot.

4. DENSITY ESTIMATION

The estimation of the intermediate result matrix densities is a crucial part of the SpMachO optimizer, since the cost models of the sparse multiplication kernels primarily depend on the number of non-zero elements, and hence, the matrix densities ρ. Our approach is to encode the non-zero structure into the smallest possible set of values without losing too much information.

a)
$$\begin{pmatrix} 0 & 2 & 0 & 0 \\ 5 & 0 & 0 & 8 \\ 0 & 0 & 8 & 0 \\ 4 & 0 & 0 & 0 \end{pmatrix} \quad \rho = \begin{pmatrix} \frac{5}{16} \end{pmatrix}$$

b)
$$\begin{pmatrix} 0 & 2 & 0 & 0 \\ 3 & 9 & 0 & 1 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 2 & 0 \end{pmatrix} \quad \begin{pmatrix} \rho_{11} & \rho_{12} \\ \rho_{21} & \rho_{22} \end{pmatrix} = \begin{pmatrix} 0.75 & 0.25 \\ 0 & 0.75 \end{pmatrix}$$

Figure 4: Assignment of non-zero population densities for two different matrices using a) scalar density, b) density map.

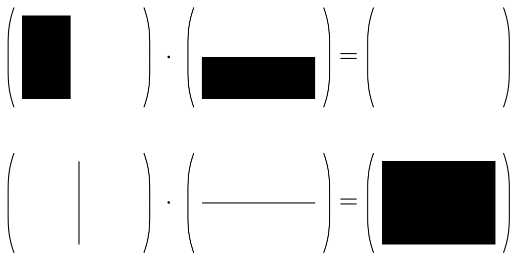


Figure 5: Product density in extreme non-uniform cases. The upper row shows how two half-populated matrices cancel to zero. The lower row shows how two, almost empty sparse matrices produce a full matrix (outer vector product).

4.1 Scalar Density

A matrix can be considered as a two-dimensional object which has a certain population density ρ of non-zero matrix elements. As an example, Fig. 4 a) shows a 4×4 matrix, which has five non-zero elements, and thus, a total population density of $\rho = 5/16 \approx 0.31$. The scalar density value does not reflect any patterns in the matrix, but it contains all relevant information if, and only if, the matrix is *uniformly* populated with non-zero elements. Then, the probability of a randomly picked matrix element for being non-zero is $p((A)_{ij} \neq 0) = \rho$, which is 0.31 in the example of Fig. 4 a).

Lemma 4.1. *Under the condition that the non-zero elements are uniformly distributed, the density estimate $\hat{\rho}$ of a product of two matrices $\boldsymbol{C} = \boldsymbol{A} \cdot \boldsymbol{B}$ can be calculated using probabilistic propagation*

$$\hat{\rho}_C = \hat{\rho}_{A \cdot B} = 1 - (1 - \rho_A \rho_B)^k. \tag{5}$$

The estimate of Eq. (5) is unbiased, i.e. $E[\hat{\rho}_C] \equiv \rho_C$.

For the sake of simplicity, we denote the operation in Eq. (5) with the symbol $\odot$, thus, $\hat{\rho}_C = \rho_A \odot \rho_B$. Lemma 4.1 can be derived as follows: using the inner product formulation, the elements c_{ij} of the result matrix are calculated as $\sum_k a_{ik} b_{kj}$. The probability for a_{ik} being non-zero is $p(a_{ik} \neq 0) = \rho_A$, for b_{kj} accordingly: $p(b_{kj} \neq 0) = \rho_B$. Thus, every summand $a_{ik} b_{kj}$ is nonzero with probability $p_{nz}(a_{ik}) \wedge p_{nz}(b_{kj}) = \rho_A \rho_B$. c_{ij} is non-zero if any of the summands $a_{ik} b_{kj}$ is non-zero. We leverage the inverse probability and obtain $p(c_{ij} = 0) = \Pi_k (1 - \rho_A \rho_B)$. Finally, with $p(c_{ij} \neq 0) = 1 - p(c_{ij} = 0)$ and $p(c_{ij} \neq 0) = \rho_C$, equation (5) results. We remark that we are assuming *no cancellation*, i.e., a sum of products of overlapping non-zero elements never cancel to zero, which is a very common assumption in the mathematical programming literature [10].

Eq. (5) can be used as an $\mathcal{O}(1)$ estimator for the result density prediction of a multiplication of two matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ that have uniform non-zero patterns. Hence, the prediction of a chain multiplication of p matrices has a linear time complexity $\mathcal{O}(p)$. Moreover, the density prediction is independent of the parenthesization.

4.2 Estimation Errors

However, the obvious disadvantage of maintaining a scalar density is that $\hat{\rho}$ is only valid for matrices with uniformly distributed non-zero elements. Although the uniform assumption holds for many matrices to a certain degree, there are many matrices that have distinguishable non-zero patterns, i.e. a topology with some regions that are significantly more dense than others. For these matrices, a density prediction according to equation 5 does not provide an accurate, unbiased result.

In extreme non-uniform cases, equation (5) could produce an asymptotic maximum error of 100%. Fig. 5 shows two example cases where the $\hat{\rho}$ estimate according to (5) fails significantly: In the first case, the product of two $n \times n$ matrices with each $\rho = 0.5$ cancel out into an empty matrix with $\rho = 0$, whereas the naive estimate according to Eq. 5 is $\hat{\rho} = 1 - (0.75)^n \xrightarrow{n \to \infty} 1$. Hence, the density value is maximal overestimated. The second example is a multiplication of two sparse $\rho = \frac{1}{n}$ matrices, which are zero except one column in $\boldsymbol{A}$ and the matching row in $\boldsymbol{B}$. The resulting full matrix has $\rho = 1$, whereas the naive prediction gives $\hat{\rho} = 1 - (1 - \frac{1}{n^2})^n \xrightarrow{n \to \infty} 0$.

In order to lower the average estimation error, we estimate matrix densities on a finer granularity. SpProdest uses a *density map* for non-uniform sparse matrices, which is able to reflect a $2D$ matrix pattern on a configurable, granular level.

4.3 Density Map

The density map $\boldsymbol{\rho}_A$ of a $m \times n$ sparse matrix $\boldsymbol{A}$ is effectively a $\frac{m}{b} \times \frac{n}{b}$ density histogram. It consists of $\left(\frac{mn}{b^2}\right)$ density values $(\rho)_{ij}$, each referring to the density of the corresponding block $\boldsymbol{A}_{ij}$ of size $b \times b$ in matrix $\boldsymbol{A}$. As an example, Fig. 4b) shows the density map of a 4×4 matrix with blocks of size 2×2.

In the following we sketch how the density map $\hat{\boldsymbol{\rho}}_C$ of the result matrix $\boldsymbol{C}$ are estimated from the density maps $\boldsymbol{\rho}_A$ and $\boldsymbol{\rho}_B$ of its factor matrices. Therefore, it is necessary to take a glance at blocked matrix multiplication. Assuming square blocks, the product matrix $\boldsymbol{C}$ can be represented as

$$\boldsymbol{C} = \begin{pmatrix} \boldsymbol{A}_{11} \cdot \boldsymbol{B}_{11} + \boldsymbol{A}_{12} \cdot \boldsymbol{B}_{21} & \boldsymbol{A}_{11} \cdot \boldsymbol{B}_{12} + \boldsymbol{A}_{12} \cdot \boldsymbol{B}_{22} \\ \boldsymbol{A}_{21} \cdot \boldsymbol{B}_{11} + \boldsymbol{A}_{22} \cdot \boldsymbol{B}_{21} & \boldsymbol{A}_{21} \cdot \boldsymbol{B}_{12} + \boldsymbol{A}_{12} \cdot \boldsymbol{B}_{22} \end{pmatrix}. \tag{6}$$

First, we define an estimator for the *addition* of two matrices:

Lemma 4.2. *Under the condition that the non-zero elements are uniformly distributed, the density estimate $\hat{\rho}$ of the addition of two matrices $\boldsymbol{C} = \boldsymbol{A} \cdot \boldsymbol{B}$ can be calculated using probabilistic propagation*

$$\hat{\rho}_{A+B} = \rho_A + \rho_B - (\rho_A \rho_B) \equiv \rho_A \oplus \rho_B. \tag{7}$$

The derivation of Lemma 4.2 is similar to that of Lemma 4.2, which is why we leave it out for space reasons.

Then, combining Eq. (6) with (5) and (7), one obtains

$$\hat{\boldsymbol{\rho}}_C = \begin{pmatrix} \rho_{A_{11}} \odot \rho_{B_{11}} \oplus \rho_{A_{12}} \odot \rho_{B_{21}} & \rho_{A_{11}} \odot \rho_{B_{12}} \oplus \rho_{A_{12}} \odot \rho_{B_{22}} \\ \rho_{A_{21}} \odot \rho_{B_{11}} \oplus \rho_{A_{22}} \odot \rho_{B_{21}} & \rho_{A_{21}} \odot \rho_{B_{12}} \oplus \rho_{A_{22}} \odot \rho_{B_{22}} \end{pmatrix}$$

for the density propagation of a 2×2 map. Density maps of a finer granularity, i.e. with more than four blocks, are calculated accordingly.

As a result, the average density estimation error is significantly lowered when using density maps compared to the scalar density estimation, which we verified empirically in the evaluation section (Fig. 6.) As a matter of fact, the smaller the block size and the higher the granularity, the more information is stored in the density map and finer structures can be resolved. However, the runtime of the density map estimation also grows with the granularity, since its complexity is in $\mathcal{O}(\left(\frac{n}{b}\right)^3)$, and hence, $\mathcal{O}(p\left(\frac{n}{b}\right)^3)$ for a chain estimation of length p. For infinitesimal block sizes $b \to 1 \times 1$, the estimation error vanishes completely, but the determination of $\hat{\boldsymbol{\rho}}_C$ is then equivalent with the corresponding boolean matrix multiplication of $\boldsymbol{A} \cdot \boldsymbol{B}$, and has the same problem complexity as the actual multiplication. Thus, the block size configuration is generally a trade-off between accuracy and runtime of the prediction.

However, we employ a greedy strategy, which reduces the runtime by using density maps only for matrices with a skewed non-zero distribution, and the scalar density for matrices with an approximately uniform distribution. To decide whether a given matrix has an uniformly distributed or a skewed non-zero pattern we define a quantitative disorder measure for sparse matrices.

4.4 Matrix Disorder Measures

We introduce two measures to quantify how the non-zero pattern of a sparse matrix deviates from an (approximate) uniform distribution. Since we already introduced the density map that involves blocks of different densities, it is natural to approach the problem from same the block-granular level.

4.4.1 *Variance Analysis*

One way of deciding whether or not a matrix is approximately uniformly distributed is to make use of a statistical hypothesis testing method. The scalar density propagation formulas as shown in Lemmas 4.1 and 4.2 are based on the assumption that every element of the matrix has the same probability to be populated, and the probability is equal to the overall density ρ. Using this assumption as the null hypothesis H_0, the number of non-zero elements in each $b \times b$-block would follow a binomial distribution $\mathcal{B}(N, p)$ with $N = b^2$ and $p = \rho$. This can be deduced in the same way as a coin toss experiment, where the number of experiments N is equal to the number of potential elements in a block, and the *success* probability $p = \rho$ equals the global population density. From elementary statistics [7] we get

$$E[N_{\text{nz}}^{b\times b}] = b^2\rho, \qquad E[V(N_{\text{nz}}^{b\times b})] = b^2\rho(1-\rho) \tag{8}$$

for the expectation values of for the number of non-zero elements N_{nz} in one $b \times b$ and the variance of $N_{nz}^{b\times b}$ when assuming a binomial distribution $\mathcal{B}(b^2, \rho)$.

The (dis-)conformance of the null hypothesis H_0 with the reality can be determined using a simple one-factorial variance analysis. Therefore, we use the F-test [7], a likelihood quotient test, which checks the conformance of the observed variance of two random, normally distributed[3] variables X and Y. If the test statistic $f = V_X/V_Y$ (ratio of variances in X and Y) exceeds a critical value f_{crit} according to an α-quantile of the F-distribution, then H_0 is rejected, meaning that X and Y are not of the same distribution with probability 1-α.

For a matrix with N_B $b \times b$ blocks, we define

$$V_{\text{observed}} = \frac{1}{N_B - 1} \sum_{ij} (N_{\text{nz}}(ij) - E[N_{nz}^{b\times b}])^2 \tag{9}$$

$$f = \left(\frac{V_{\text{observed}}}{V_{\text{expected}}}\right) = \left(\frac{V_{\text{observed}}}{E[V(N_{nz}^{b\times b})]}\right) \tag{10}$$

For uniformly distributed matrices, f has the expectation value $E[f] = 1$. The exact choice of the threshold, however, depends on the sample size, i.e. the number of blocks N_B and the desired accuracy.

4.4.2 *Entropy*

A known measure for the disorder of elements is the entropy

$$\sum_i^N -p_i \ln p_i, \tag{11}$$

which is defined over a space with N entities (or states) i that have a relative probability p_i. The entropy is used in a variety of contexts. To name an example, the Shannon entropy [26] is used in information theory to quantify the information content of a message with N characters as entities. In a similar manner, we can define and quantify the information content of the non-zero pattern of a sparse matrix, by identifying the p_i with the local block density ρ_{ij}.

The entropy (11) is maximal if each entity has the same probability. In the terms of sparse matrices, the theoretical disorder is maximized if every block has the same local density $\rho_{ij} \equiv \rho$. The scaled entropy

$$\tilde{H} = \frac{H}{H_{\max}} = \frac{\sum_{ij}^{N_B} \rho_{ij} \ln \rho_{ij}}{N_B \rho \ln \rho} \in [0, 1] \tag{12}$$

is sensitive to the matrix density skew, which we evaluated in section 6. However, in contrast to f, the entropy is rather suited for measuring *relative* changes in the non-zero disorder, and it is hard to interpret the absolute value of $\tilde{H}$.

5. SpProdest

SpProdest is sketched in algorithm 2. First, the disorder measure $\delta = \sqrt{1/f}$ is retrieved (getDisorder) for each matrix, which is a modified version of f according to Eq. (10). Then, δ is used to decide whether to store a only scalar density value, or a density map (line 4). If δ is lower than a certain threshold δ_T, then the density map is created, if the disorder is higher, then a scalar density is chosen. Finally, the density estimates are calculated by using the probabilistic density propagation method (EstProdDensity), according to equations (5) and (7). Note that instead of calculating δ and $\hat{\boldsymbol{\rho}}$ for each expression (line 4-7), they can be cached as matrix statistics in the system, and reused for further multiplications.

The complexity of SpProdest depends on the actual granularity of the density map. Assuming a chain multiplication

[3]for sufficiently large N and $Np \to$ const., the binomial distribution can be approximated by a normal distribution

Algorithm 2 SpProdest

```
1: function spProdest(MatrixChain $\boldsymbol{A}_{[1..p]}$)
2:     $\hat{\boldsymbol{\rho}}[][] \leftarrow 0$
3:     for Matrix $\boldsymbol{A}_i \in \boldsymbol{A}_{[1..p]}$ do
4:         $\delta \leftarrow$ getDisorder($\boldsymbol{A}_i$)
5:         if $\delta < \delta_T$ then
6:             $\hat{\boldsymbol{\rho}}[i][i] \leftarrow$ scalarDensity($\boldsymbol{A}_i$)
7:         else
8:             $\hat{\boldsymbol{\rho}}[i][i] \leftarrow$ densityMap($\boldsymbol{A}_i$)
9:     for $1 < j < p$ do
10:        for $j > i > 0$ do
11:            $\hat{\boldsymbol{\rho}}[i][j] \leftarrow$ estProdDensity($\hat{\boldsymbol{\rho}}[i][j-1], \hat{\boldsymbol{\rho}}[j][j]$)
       return $\hat{\boldsymbol{\rho}}[][]$
```

of p square matrices, the time complexity would be in the best case $\mathcal{O}(p)$ (no map) and in worst case $\mathcal{O}(p(\frac{n}{b})^3)$, which is equal to the chain multiplication of p $\frac{n}{b} \times \frac{n}{b}$ matrices, where $\frac{n}{b}$ is the dimension of the density grids. Analogously, the space complexity of SpProdest is best case $\mathcal{O}(p)$, worst case $\mathcal{O}(p(\frac{n}{b})^2)$. In practice, the overhead of the SpProdest component is negligible against the potential speedup gained by the SpMachO, which is manifested in our evaluation in section 6.3.

6. EVALUATION

In this section, we first evaluate the accuracy of spProdest according to the deviation in the result sparse matrix densities. Second, we apply SpMachO on different matrix chains and compare the execution runtime against R and a popular commercial numerical algebra system for matrix computations (called system A here), and two further execution approaches. The platform for our prototype implementation is a two-socket Intel Xeon X5650 CPU with 2 × 6 cores with 2.66 GHz and a total of 48 GB RAM.

6.1 Density Estimate Accuracy

As mentioned in section 4, the density ρ_C of a matrix $\boldsymbol{C} = \boldsymbol{A} \cdot \boldsymbol{B}$ depends not only on the densities of the factor matrices ρ_A and ρ_B, but also on their non-zero patterns, especially on the pattern skew. To show the effect of the non-zero pattern skew on the density estimation, we generated a set of matrices with increasing skew. The skew parameter ξ in our example defines the slope of a linear ascend in the density distribution with increasing row number r: $\rho(r) = \xi \cdot r$. We fixed the total density of $\boldsymbol{A}$, thus, N_{nz} =const.

The left-hand plot in Fig. 6 shows the actual density ρ_C and the estimated densities using the scalar density and the density map approach with different block sizes. In our example, the density of the product matrix ρ_C decreases with increasing pattern skew. This conforms with our notion, that in most cases a higher skew at constant density leads to a lower density of the result matrix, although there are cases which show the opposite behavior, for example, as in Fig. 5.

Nevertheless, it is clearly observable that a finer granularity of the density map, and thus, a smaller block size, results in a better density estimation. The idea is to choose the block size as large as possible, since a finer granular density grid negatively influences the runtime performance of spProdest. We chose a block size of 256 × 256 as a good compromise between accuracy and estimation runtime. The right-hand plot of Fig. 6 confirms that the disorder measures are both

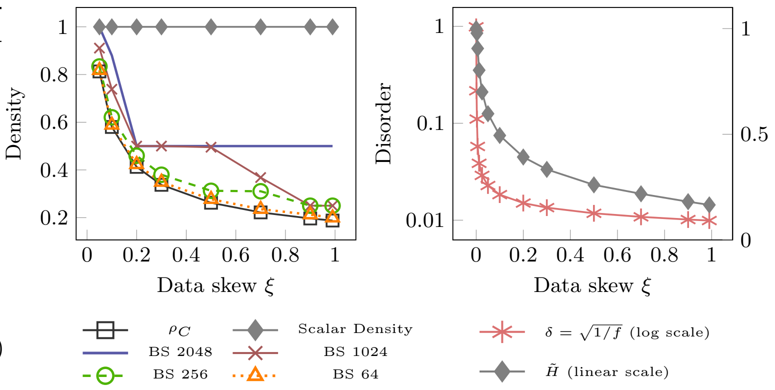


Figure 6: Left: Estimated density $\hat{\rho}_C$ vs. actual density (ρ_C) of the product matrix $\boldsymbol{C} = \boldsymbol{A} \cdot \boldsymbol{B}$ using the scalar and the density map estimation with different grid block sizes (BS). Matrices: $\boldsymbol{A} \in \mathcal{R}^{4096 \times 2048}, \boldsymbol{B} \in \mathcal{R}^{2048 \times 4096}$ and average density $\langle\rho\rangle = 0.1$. Right: Influence of the matrix $\boldsymbol{A}, \boldsymbol{B}$ nonzero skew on the disorder measures $\delta, \tilde{H}$.

sensitive to the nonzero skew, but the teststatistic-based disorder measure δ is much more sensitive to the skew than the entropy-based $\tilde{H}$. In particular we can observe that the trivial scalar density provides a sufficient accuracy for approximately uniform nonzero patterns ($\xi \to 0$).

6.2 Plan Ranking

In this experiment we evaluate the total cost model accuracy of SpMachO by comparing the estimated runtime against the actual runtime of each possible plan. Hence, SpMachO is optimal if the plan with the lowest actual runtime has also the lowest estimated runtime. However, this experimental verification of optimality requires to run all possible execution plans, which is not feasible for longer matrix chains due to the exponentially growing number of plans according to Eq. (4). Thus, we did a "brute-force evaluation" only for matrix chains of length $p = 3$:

$$\boldsymbol{A}_1^S \cdot \boldsymbol{A}_2^S \cdot \boldsymbol{A}_3^S. \tag{13}$$

Although there are only two ways of setting the parenthesis in this expression, with all the possible storage type transformations per multiplication node, one obtains 128 different execution plans. Fig. 2 shows some of the possible plans, to illustrate the problem complexity. The execution plans are composed of

- **multiplication operators** ${}^{D/S}\times^{D/S}_{D/S}$, which can either produce a *sparse* result matrix or a *dense* one. For a chain of length p there are exactly $p-1$ multiplication operators.
- **transformation operator** $T^{S/D}_{S/D}$ that transforms the intermediate result from one storage representation into another (which is either dense or sparse.) There can be none or up to 2^{3p-1} transformation operators.

SpMachO estimates the cost for each operator via the cost functions `TM`,`TT` described in section 2. As a consequence, each operator estimation potentially contributes to the absolute execution runtime error.

In this experiment, we first executed all 128 plans and measured the *actual* execution runtime. Thereafter, we computed the runtime estimations using SpMachO's cost model

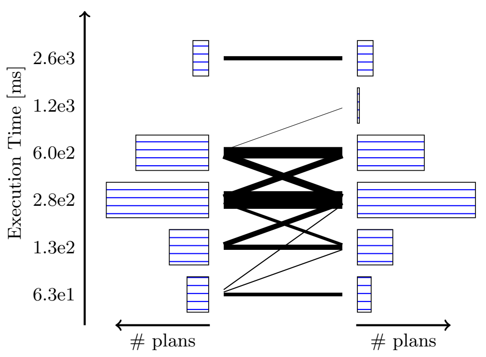


Figure 7: Vertical histogram of the actual (bars on left-hand side) and estimated (bars on right-hand side) runtimes of all 128 possible execution plans. The edges denote the plan (dis-)placement, the linewidth correlates with the corresponding number of plans. The vertical time axis has a logarithmic scale.

for each plan. Fig. 7 shows the histograms for the actually measured (left-hand side) and the estimated (right-hand side) execution runtimes. Note that the vertical axis has a logarithmic scale, hence, the width of the upper bins refers to a larger time interval than the width of the lower bins. The connecting edges in-between the bins of the two histograms show where the plans of the actual runtime histogram are placed in the estimated runtime histogram. If there is an ascending edge, for example from the lowest bin in the left histogram to the second lowest bin of the right histogram, then there is at least one plan, whose runtime was overestimated. If the edge is horizontal, then the estimated runtimes of all plans corresponding to this edge are within the same time interval as their actual runtimes. The width of the edges indicates how many plans are affected. It is worthwhile mentioning that for the selection of the best execution plan, the quantitative estimation of execution runtimes could potentially differ arbitrarily from the actual runtimes, as long as the estimated *order* of the plans preserves the actual runtime order correctly. This condition is only violated for the edges that are crossing another edge. If the total runtime for a plan is significantly under- or overestimated, its corresponding edge crosses multiple other edges. Indeed, the goodness of the cost model can be defined by the number of edges crossings, weighted by the number of plans per crossing edge.

The majority of estimations, which are shown in Fig. 7, are in the correct corresponding time bin. Although there are quite a few crossings, especially in the middle part, most of the edges only span over to the neighboring bin. Moreover, for the selection of the most efficient plan, only the lower part of Fig. 7 is relevant. In particular, the plan with the lowest estimated runtime, which is generated by SpMachO, should be contained in the lowest bin of the actual runtime histogram. Since all edges of the lowest estimated time bin originate from the lowest actual time bin, we observe that the SpMachO selected plan is at least among the top k plans, if not the best.

6.3 Performance Comparison

We compared the absolute execution runtime of a matrix chain multiplication expression using the optimized plan by SpMachO against R and the commercial system A. Both systems contain classes and algorithms for dense and sparse matrices. In R (V3.0.0) we used the Cran R matrix package[2] (V1.0.12), in system A we used the native sparse matrix representation. In addition,we included the following alternative execution approaches to the measurement:

- **Left-deep, sparse only:** All matrices are multiplied using a sparse-sparse into sparse multiplication using the *spspsp_gemm* kernel (${}^{S}\times^{S}_{S}$), starting with the first left pair and proceeding into the right direction.
- **Right-deep, sparse-dense-dense:** The outermost right pair is multiplied using sparse-sparse into dense multiplication (${}^{S}\times^{S}_{D}$). Then, the matrices on the left are consecutively multiplied with the right-hand dense intermediate result matrix using the sparse-dense into dense multiplication kernel (*spdd_gemm*, ${}^{S}\times^{D}_{D}$).

Both approaches use the same infrastructure as SpMachO. The reason why we chose exactly these two specific execution strategies is that either of them turned out to be good (or even optimal) for a reasonable large fraction of matrix chains. In particular, they yield good performance if the inter-matrix skew is low, i.e., the dimensions and the densities do not differ, since in these cases, the impact of parenthesization on the optimization is less significant. We also tried other alternatives to the dynamic programming approach of SpMachO, for example, a method that picks an execution plan based on the metaheuristic simulated annealing. However, due to the high dimensionality of the search space and the large discrepancy in the runtimes, it turned to be out to be far worse in most cases, hence, we did not include it in the measurements.

6.3.1 Data Set

Since there are currently no standardized benchmarks for large scale linear algebra expressions, it is generally difficult to provide a comprehensive performance comparison. Therefore, we created two performance experiments: first, we took real world-matrices of different domains and compared the execution runtime of self-multiplication chains (matrix powers). Thereafter, we generated random matrices of different dimension and density skews in order to study the systematic behavior of SpMachO.

Table 2: Sparse matrices of different dimensions and population densities. The $\rho = N_{nz}/(n \times n)$ value denotes the population density (rounded) of each matrix. All matrices are square ($n \times n$.)

Name	Matrix Domain	Dim.	N_{nz}	$\rho \cdot 10^2$ [%]
NCSM1	Nuclear Physics	3440	2.930 M	24.7
PWNET	Power Eng.	8140	2.017 M	3.0
JACO1	Econometric	9129	56 K	0.07

Tab. 2 lists the matrix data sets which we used in the evaluation. The first matrix NCSM1 is taken from a nuclear physics group, the other two (PWNET, JACO1) are from the Florida Sparse Matrix Collection[4].

In our prototype system, some of the multiplication kernels are implemented single-threaded, whereas other kernels have parallel implementations. Although we emphasize that the

[4]`http://www.cise.ufl.edu/research/sparse/matrices/`

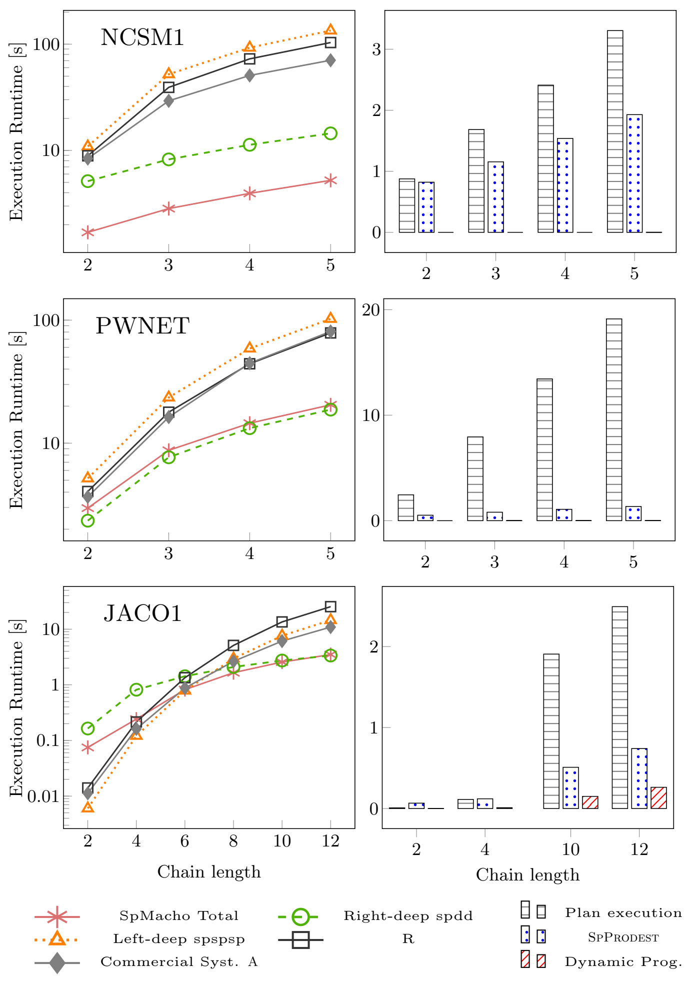


Figure 8: *Left Column*: Measurement of the execution runtime of sparse matrix chains (matrix powers). *Right*: The total runtime of SpMacho is visually separated into its components: the plan execution, the SpProdest runtime and the dynamic programming part.

conceptual execution plan optimization of SpMacho works orthogonal to the individual performance of the multiplication kernels, the absolute execution runtime does obviously also depend on the low level implementation of each algorithm. Hence, there is still potential to reduce the overall runtime further by switching completely to massively parallelized multiplication kernels. However, although we use mostly sequential kernels in the prototype, our algorithm was still able to outperform R and the commercial system A.

6.3.2 Self Multiplications

In this part we discuss the performance of matrix self multiplications (matrix powers), which is for example used for the calculation of Markov chains models.

The left column of Fig. 8 shows the absolute runtimes using SpMacho versus R, commercial system A, and the right-deep *spspsp* and left-deep *spdd* approaches. The first notion is that the relative performance speedup of SpMacho becomes more significant with increasing chain length. For matrices with a relatively high density, e.g. NCSM1, SpMacho outperforms the other systems even for a single multiplication already by several factors. In this case, SpMacho recognizes that it is worth to convert the matrix into dense representations prior to the multiplication. For the second matrix chain (PWNET), the performance gap is increasing with the chain length up to a speedup factor of five. Only in the third plot (JACO1), the overhead of SpMacho amortizes not before a chain length of four. In this relatively simple case of matrix self multiplications, the speedup is related to the density evolution of the intermediate results. When the matrix reaches a relatively high density in an early stage of the execution plan, SpMacho is likely to choose dense formats and proceed with dense multiplication kernels, whereas the use of sparse-only kernels will have a poor performance for every additional multiplication. That also explains why the right-deep *spdd_gemm* strategy is often optimal, e.g. for the PWNET matrix chain.

The right column in Fig. 8 shows the separate runtimes of each component of SpMacho, which are: the plan execution, the SpProdest runtime, and the dynamic programming loop. The runtime of the dynamic programming part is negligible and only visible for longer chains (10-12). Note that we the SpProdest cost can be further reduced, if we cache the density maps. As of now, they are created once prior to each expression execution, consuming most of SpProdest's runtime.

As a side note, the R and commercial system A runtimes show astonishing similarity with our left-deep *spspsp_gemm* approach. We assume that they use a similar way of execution.

6.3.3 Random Matrix Chains

In the next experiment, we used three different, randomly created matrices. Products of *three* matrices are very common in many applications, for example in algorithms that contain matrix factorizations.

In order to observe the systematic influence of the data skew on the execution runtime, we varied three skew dimensions: the matrix shape skew, the inter-matrix density skew and the matrix intra-density skew (as of section 6.1). For each skew dimension, we varied a parameter $\xi \in [0, 1]$ that quantifies the skew in a range from zero (no skew) to one (maximum skew). More precisely, the parameter dimensions $(m/n)_i$, ρ_i and the intra-density skews ξ_i, are randomly picked from a <*min*, *max*, *average*> distribution, where ξ corresponds to the deviation from the *average* value.

Since we created the matrices randomly for each skew parameter configuration, one single configuration can have various random instances, which results in a potentially large variety of different runtimes. This is reasoned by the fact that a skew in the matrices can affect the execution runtime in both directions – increasingly or decreasingly. Generally speaking, a large skew in the data can dramatically slow down naive execution approaches, but also reveals a large optimization potential for SpMacho by exploiting the skew. In contrast to the previous self-multiplication experiment, a skew in the matrices leads to a higher influence of the parenthesization, and the selection of storage representations and algorithms.

To be independent of particular random matrix instances, we repeated the measurement multiple times, hence, we took 25 different randomly created sparse matrix chains of length three per configuration. Fig. 9 shows for each skew configuration a box plot with the corresponding median, lower quartile,

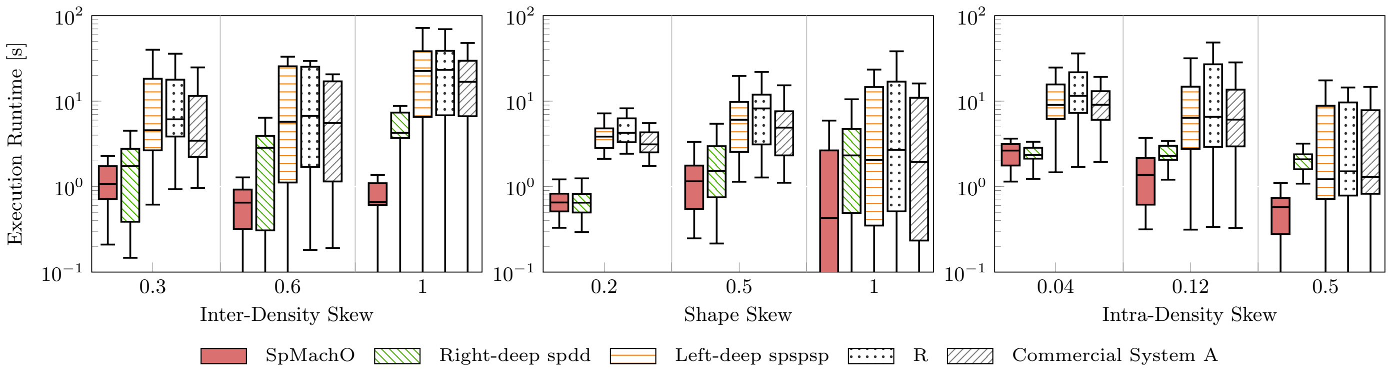


Figure 9: Average (log scale) runtime and variance comparison of SpMachO vs. the right-deep spdd and left-deep spspsp approaches, R, and the commercial system for a multiplication of the expression $\boldsymbol{A}_1 \cdot \boldsymbol{A}_2 \cdot \boldsymbol{A}_3$. We varied in x-direction: the inter-density skew (left), the shape skew (middle) and the intra-density skew (right). The bounding *<min,max,avg>* distributions for the data skew are *<0.001, 0.5, 0.025>* for matrix densities, and *<32, 16384, 3072>* for the matrix dimensions. The intra-density skew ξ was chosen according to Fig. 6 with values ranging from 0 to 0.5.

upper quartile and whiskers of the execution runtimes. Note that our measured runtime of SpMachO includes the time for the density estimation (SpProdest). We observe the following characteristics: First, for low skew parameters, both SpMachO and the right-deep spdd outperform the other approaches for most of the instances. For unskewed matrices, this underlines the result that we already obtained in the previous experiment for matrix chains with similar densities, i.e. that the right-deep spdd multiplication execution is optimal if intermediate results are rather dense. Second, and more interesting, is the development of the runtime distributions with higher data skews. In the inter-density skew experiment (left plot), the median execution time in R, the commercial system A and the left-deep spspsp approach increases, whereas the SpMachO median time stays low and gets even lower for $\xi = 1$. Moreover, the variance in time of SpMachO grows notably slower than these of the other systems. In the other two plots, we see a similar picture, although most of the median execution times decrease slightly in the shape skew experiment (middle plot), and more significantly in the intra-density skew experiment (right plot). Here, the increased intra-density skew leads in the majority of cases to a reduced execution runtime, which conforms to our notion. Still, in quite a few cases the runtime of the other systems explodes, leading to the observed high variance and scattering of the execution times. The right-deep spdd approach is more robust, but not optimal for high skews. In contrast, SpMachO is able to reveal the skew and exploit it for optimization. As a result, we observe that SpMachO has a by far better worst-case behavior than the established systems.

7. RELATED WORK

As this work has overlaps with multiple research areas, we subdivide the discussion into the major subtopics:

7.1 Optimization of Linear Algebra Expressions

Despite the optimization potential, we did not find that common numerical algebra systems optimize the execution of linear algebra expressions based on matrix sparsity and dimension characteristics. In contrast, the idea of optimizing linear algebra operations on system level has been mentioned in SystemML [14, 6], which describes a Hadoop-based machine-learning framework with an *R*-like declarative language. However, the cost model they describe in [6] is based on independent, one-dimensional scaling functions, and they assume full density ($\rho = 1$) for intermediate results. In [5] they mention that they optimize by assuming “independence with regard to the sparsity of intermediates”. In contrast, we observed that particularly in situations with large density differences (inter-density skew) the density of intermediate results influences the optimization significantly. Since simple models are unable to reconstruct the complicated runtime behaviour of matrix multiplication kernels, we also put our focus on accurate cost models from an algorithmic perspective. In addition, our cost analysis revealed that matrix dimensions and the matrix sparsity can not be regarded as independent parameters.

7.2 Matrix Chain Multiplication and Density Estimation

In contrast to dense matrix chain multiplication, which has been discussed thoroughly in the past decades, e.g. in [12, 16, 21], there is little work about sparse matrix chain multiplications. Interesting work that should be mentioned in this context is from Cohen [9, 10], who extended the dynamic programming approach idea to sparse matrices. In her work, she minimizes the overall number of floating point operations that are needed to compute the matrix chain product by predicting the non-zero structure for intermediate result matrices on row/column-level. The density prediction algorithm of [10] is based on random number propagation in a layered graph, and has a fixed complexity $\Theta(\sum N_{nz,i})$. However, as observed in section 2, the actual runtime cost of sparse matrix multiplication kernels are not just proportional to number of floating point operations. Moreover, coming from a real system perspective, we consider not only pure sparse-sparse matrix multiplications, but leverage sparse-dense transformations and the coexistence of sparse and dense matrices to optimize on a more complete level.

7.3 Join Optimization

The problem of sparse matrix chain multiplication is related to join enumeration and cardinality estimation in a relational database management system RDBMS. This connection is more obvious when sparse matrices are represented as $\langle row, col, val \rangle$ triple tables [20]. In fact, a multiplication can then be expressed as a join aggregation [3]. The use of dynamic programming in join optimization [25, 22] and join plan generation with respect to physical table properties [17] were inspiring for this work, as well as the use of multidimensional histograms [23] for the optimization of queries on multidimensional data. Although the mathematical characteristics of matrices and multiplications require a slightly different perspective, it is an interesting aspect that some of the ideas of relational join optimization can be used for linear algebra.

8. CONCLUSION

In times of emerging analytical and scientific databases, many systems [8, 6] started to deeply integrate linear algebra. This work shows that integrating linear algebra operations, such as matrix multiplications, is not just adding algorithms to the database engine. In fact, due to different matrix representations, algorithms, and the presence of data skew, we observed that a naive execution of sparse matrix products can be up to orders of magnitude slower than an optimized one.

In this paper we presented SpMachO, which optimizes sparse, dense and mixed matrix multiplications of arbitrary length, by creating an execution plan that consists of transformation and multiplication operators. By using detailed cost functions of different sparse, dense and mixed matrix multiplication kernels, SpMachO leads to a faster and more robust execution compared to widely used algebra systems. Moreover, our density prediction approach SpProdest with an entropy-based skew awareness enables accurate memory consumption and runtime estimates at each stage in the execution plan.

To put it in a nutshell, we showed how methods inspired from database technology can improve linear algebra computations, and took a step into the direction of taking complexity from data scientists – who should not be required to have profound knowledge about the connections between mathematical optimizations, matrix characteristics, algorithmic complexities and the hardware parameters of their system.

9. REFERENCES

[1] Intel® Math Kernel Library, http://software.intel.com/en-us/intel-mkl.

[2] CRAN R Matrix Package, http://cran.r-project.org/web/packages/Matrix/index.html.

[3] R. R. Amossen and R. Pagh. Faster Join-Projects and Sparse Matrix Multiplications. In *ICDT*, 2009.

[4] K. Behrend. Dynamical Systems and Matrix Algebra. 2008.

[5] M. Boehm, R. B. Douglas, A. V. Evfimievski, et al. SystemML's Optimizer: Plan Generation for Large-Scale Machine Learning Programs. *IEEE Data Eng. Bull.*, 37(3), 2014.

[6] M. Boehm, S. Tatikonda, B. Reinwald, et al. Hybrid Parallelization Strategies for Large-Scale Machine Learning in SystemML. *VLDB*, 7(7), 2014.

[7] G. E. P. Box, J. S. Hunter, and W. G. Hunter. *Statistics for Experimenters: Design, Innovation, and Discovery , 2nd Edition*. Wiley-Interscience, 2 edition, May 2005.

[8] P. G. Brown. Overview of SciDB: Large Scale Array Storage, Processing and Analysis. In *SIGMOD*, 2010.

[9] E. Cohen. On Optimizing Multiplications of Sparse Matrices. In *IPCO*, 1996.

[10] E. Cohen. Structure Prediction and Computation of Sparse Matrix Products. *Journal of Combinatorial Optimization*, 2(4), 1998.

[11] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, et al. MAD Skills: New Analysis Practices for Big Data. *VLDB*, 2(2), Aug. 2009.

[12] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[13] A. Edelman, S. Heller, and S. Lennart Johnsson. Index Transformation Algorithms in a Linear Algebra Framework. *IEEE Trans. Parallel Distrib. Syst.*, 5(12), Dec 1994.

[14] A. Ghoting, R. Krishnamurthy, E. Pednault, et al. SystemML: Declarative Machine Learning on MapReduce. In *ICDE*, 2011.

[15] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse Matrices in Matlab: Design and Implementation. *SIAM J. Matrix Anal. Appl.*, 13(1), Jan. 1992.

[16] S. S. Godbole. On Efficient Computation of Matrix Chain Products. *IEEE Trans. Comput.*, 22(9), Sept. 1973.

[17] G. Graefe and W. J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proc. Int. Conf. Data Eng.*, 1993.

[18] F. G. Gustavson. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. *ACM Trans. Math. Softw.*, 4(3), Sept. 1978.

[19] IBM® Netezza® Analytics. *Matrix Engine Developer's Guide*. IBM, 1994.

[20] D. Kernert, F. Köhler, and W. Lehner. SLACID - Sparse Linear Algebra in a Column-oriented In-memory Database System. In *SSDBM*, 2014.

[21] H. Lee, J. Kim, S. J. Hong, and S. Lee. Processor Allocation and Task Scheduling of Matrix Chain Products on Parallel Systems. *IEEE Trans. Parallel Distrib. Syst.*, 14(4), Apr. 2003.

[22] G. Moerkotte. Constructing Optimal Bushy Trees Possibly Containing Cross Products for Order Preserving Joins is in P. 2003.

[23] M. Muralikrishna and D. J. DeWitt. Equi-depth Multidimensional Histograms. *SIGMOD Rec.*, 17(3), June 1988.

[24] R. Rebonato and P. Jäckel. The Most General Methodology to Create a Valid Correlation Matrix for Risk Management and Option Pricing Purposes, 1999.

[25] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *SIGMOD*, 1979.

[26] C. E. Shannon. A mathematical theory of communication. *SIGMOBILE Mob. Comput. Commun. Rev.*, 5(1), Jan. 2001.

[27] V. Yegnanarayanan. An application of matrix multiplication. *Resonance*, 18(4), 2013.

Efficient evaluation of threshold queries of derived fields in a numerical simulation database

Kalin Kanov
Department of Computer Science
IDIES
Johns Hopkins University
Baltimore, MD 21218
kalin@cs.jhu.edu

Randal Burns
Department of Computer Science
IDIES
Johns Hopkins University
Baltimore, MD 21218
randal@cs.jhu.edu

Cristian C. Lalescu
Department of Applied Mathematics and Statistics
IDIES
Johns Hopkins University
Baltimore, MD 21218
clalesc1@jhu.edu

ABSTRACT

In this paper, we present a method for the efficient evaluation of threshold queries of derived fields for large numerical simulation datasets stored in a cluster of relational databases. The datasets produced by these simulations are in the TB and even PB ranges. Data-intensive computations that examine entire time-steps of the simulation data are impractical to perform locally by the user, taking days or months to iterate over the entire dataset. The integrated method for the evaluation of threshold queries that we have developed achieves scalability through data-parallel execution of the computations on the nodes of an analysis database cluster. We extend the scientific analysis environment with the introduction of an application-aware cache for query results, building on the concept of semantic caching. The cache has little overhead and improves query performance by over an order of magnitude for queries that hit the cache. Caching the results of threshold queries preserves both the I/O and computation effort used to obtain them. In the case of computational turbulence, this allows scientists to quickly focus on the most intense events and interesting regions in any time-step or the dataset as a whole, which greatly speeds up the rate of scientific exploration and discovery.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications – Scientific Databases; H.2.4 [**Database Management**]: Systems – Distributed Databases; J.2 [**Computer Applications**]: Physical Sciences and Engineering – Physics

Keywords

scientific databases, data-intensive computing, threshold queries, turbulence

1. INTRODUCTION

Better instruments, faster and bigger supercomputers and easier collaboration and sharing of data in the sciences have introduced the need to manage increasingly large datasets. Data-intensive systems and architectures have been developed with the goal of storing and providing fast access to such datasets. Examples of such analysis environments include the GrayWulf and Data-Scope clusters [31, 10] at Johns Hopkins, which have capacity of 1.1PB and 11PB respectively. One of their missions is to provide persistent storage and public access to world-class numerical simulation data. These systems differ from the traditional HPC environments in that they aim to achieve high aggregate throughput by balancing computation capabilities with I/O and network bandwidth. The computing systems and services developed on top of these platforms are more than pure storage engines and usually have complex analysis routines built-in, which has largely been driven by the "move the computation to the data" paradigm [14]. These built-in analysis routines are most often not novel themselves. They implement core scientific functionality for the study of the particular scientific phenomena, which was observed or simulated in the first place. The analysis routines however require novel evaluation strategies and methods for their execution. They have to operate on large array datasets distributed across multiple nodes of a cluster of relational databases. In order to reduce their running times, they have to make efficient use of the cluster resources and incorporate leading data management techniques.

Finding the locations or regions of highest vorticity or those with the largest norms of the velocity or other fields of interest enables new insights in the study of fluid dynamics. Analysis of this kind coupled with the ability to analyze time-series datasets both forward and backward in time has transformed our understanding of turbulence [12]. Furthermore, threshold, top-k queries and similarity search in general are important in many different disciplines. We introduce an efficient evaluation strategy for threshold queries over time-series datasets stored in a cluster of relational databases. Our method evaluates not only threshold queries of the vector or scalar field data stored in the database, but also performs thresholding of *derived* fields.

The main challenge that our approach tackles is that the data-intensive computation of *derived* fields has to be carried out on-demand for extremely large array datasets stored in an analysis cluster environment comprised of multiple

10.5441/002/edbt.2015.27

database nodes. We focus on the evaluation of threshold queries of fields derived from the data stored in the cluster as these queries are the most interesting scientifically. However, our approach applies to the evaluation of top-k queries, rollup queries and data-reducing queries in general. The established data management techniques that our approach combines make the approach easy to understand. It can be applied to other scientific analysis environments, which manage large datasets in a database management system. Examples include the Sloan Digital Sky Survey [28], the Millennium Simulation [23] and the Open Connectome Project [6].

Evaluating threshold queries within the database cluster allows scientists with modest computational and network capability to narrow down on and examine some of the most interesting regions and features in the dataset and focus on the subsequent analysis needed to understand these events. It is impractical to materialize all possible derived fields and store them alongside the raw data due to the large size of the datasets and the limits of available storage. Obtaining the derived field and thresholding locally by the user requires not only the computation of the derived field over an entire time-step server-side but also the transfer of a large amount of data over the network, most of which are subsequently discarded. One of our collaborators reported that such a local evaluation of a threshold query over an entire time-step took over 20 hours. It would take months to iterate over the entire dataset. This highlighted the need for providing the capability through an integrated approach, which performs the evaluation server-side.

Database, operating and file system caches are effective at speeding up access to the large amounts of data stored on disk. However, this might not be sufficient for some applications, because these application-independent caches cannot exploit dataset-specific structure and application-level information [20]. Moreover, even if the data are available in one or more of these application-independent caches the computation associated with the derived field still needs to be performed for each point on the grid, because results of previous computations are not cached. We will demonstrate that an integrated approach, which computes the derived fields on-demand in a data-parallel manner, performs the evaluation over an entire time-step in a few minutes. Storing the query results in an application-aware semantic cache further reduces the running times to several seconds.

Thresholding allows scientists to obtain and examine the regions containing the most intense events and features in the dataset in the case of turbulence. These are often the locations that have the largest vorticity norms and have intense vortices or reconnection events. In magnetohydrodynamics, the locations of largest electric current are of great interest for similar reasons. It is important that threshold queries are evaluated in an efficient manner, because often further subsequent examination and analysis is required to understand the physics that drive these intense events.

There are several challenges that arise during the evaluation of threshold queries of derived fields in an analysis database cluster. The field variables have to be evaluated *on-demand* from the array data stored in the database cluster. The evaluations are data-intensive as they perform kernel computations on extremely large multidimensional array datasets. A kernel computation computes the value at a grid location using the data points at a set of neighboring locations. Kernel computations have to be performed at each location on the grid as opposed to at a particular number of target locations. The evaluation needs to be distributed across the nodes of the database cluster to avoid the unnecessary movement of data over the network and to achieve scalability. Techniques that target the traditional supercomputing environments do not translate directly to the distributed database setting of an analysis cluster environment.

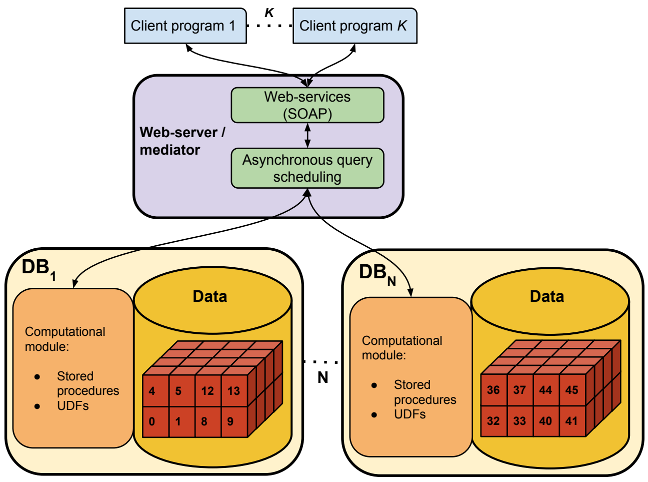


Figure 1: Architecture of the JHTDB.

We present a method for the efficient evaluation of threshold queries over fields derived from the raw vector or scalar fields of the numerical simulation stored in the database. Our method makes effective use of the cluster resources and achieves high throughput and scalability. We exploit the parallelism available in the cluster by means of data-parallel execution of the computations. We extend the database management system with an application-aware cache for query results. We build on the idea of an application-aware cache introduced by Lopez et al. [20] and more broadly on the concept of semantic caching [9]. Rather than caching just data as is the case in system caches and the tree-cache described by Lopez et al., we cache query results along with query metadata and subsequent queries are evaluated against the cache. This leads to query performance improvement of over an order of magnitude.

The contributions of this paper are the following:

- Computing derived fields of large simulation data on-demand and evaluating threshold queries on them at extreme scale. This provides large data analytics capabilities that examine entire time-steps of the simulation transparently to the user in a production analysis environment.
- Achieving this through the combination of existing data management techniques such as data parallelism and semantic caching as well as taking advantage of heterogeneous scientific cluster architectures (sharded relational DBMS with several SSDs per node).
- Evaluating the proposed method on data-intensive workloads in a live production environment and showing scalability results on datasets hundreds of terabytes in size.

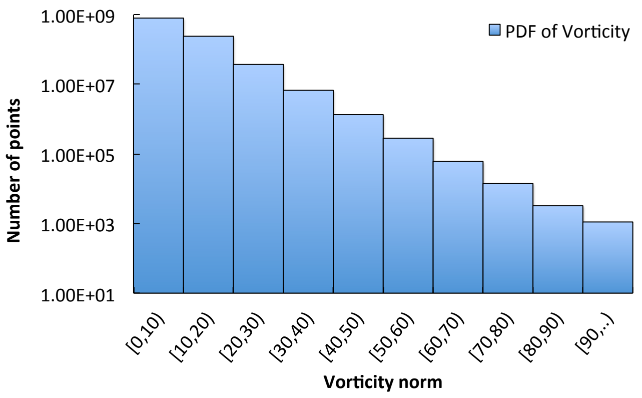


Figure 2: Probability density function of the norm of the vorticity field for a representative time-step for the MHD dataset.

2. JOHNS HOPKINS TURBULENCE DATABASES

Data-intensive architectures and compute clusters built from commodity hardware rely on parallel I/O to multiple disks and high network bandwidth to achieve high throughput. Such systems have only recently been deployed for the storage of large numerical simulation datasets. The virtual laboratories built on these systems make use of relational database system technology to store and manage large array datasets. Relational database systems however often do not support all of the functionality that scientists are interested in out of the box. It is either up to the user to develop more sophisticated analysis routines locally or such capabilities have to be built into the database through user-defined functions or stored procedures.

The method that we have developed for the evaluation of threshold queries of derived fields was deployed and integrated into the Johns Hopkins Turbulence Databases (JHTDB) [19, 26]. It solves a pressing problem in a production scientific analysis environment, which differs from the traditional supercomputing environments and provides large data analytics capabilities transparently to the public. The JHTDB, built on top of the GrayWulf and Data-Scope clusters, serves as a public virtual laboratory for the study of turbulent phenomena. The JHTDB stores several datasets, which are the output of high-resolution numerical simulations of turbulence. The 3d time-series data are partitioned into small subcubes and stored in relational databases distributed across the nodes of the cluster. Access to the data is provided by means of Web-services and a variety of analysis functions have been implemented and can be executed through Web-service calls (Fig. 1). At present the service hosts four datasets, which are available publicly. The data are the output of numerical simulations of forced isotropic turbulence, magnetohydrodynamics (MHD), channel flow turbulence and homogenous buoyancy driven turbulence. The total amount of space occupied by the datasets is over 230 TB.

The database nodes are part of the GrayWulf [31] and Data-Scope [10] clusters. Each node is running Windows Server 2008 and SQL Server 2008 R2. The data for each dataset reside on a regular three-dimensional spatial grid with the exception of the channel flow data, which has an irregular y dimension. The data are partitioned spatially across 4 to 8 database nodes, and each database node hosts one or more databases. We use the Morton z-order space-filling curve to distribute the data across nodes and databases [26]. Each time-step is spatially subdivided into database atoms, which are of size 8^3. Each such atom is indexed by the time-step, which it belongs to and by the Morton code of it's lower left corner. This combination of index and data forms a record in the database. Queries to the data and derived fields, such as derivatives and filtered quantities are evaluated through stored procedures or user-defined functions implemented in the Common Language Runtime (CLR) framework.

The Web-services are hosted on a front-end Web-server, which handles user requests and hosts the main Web-page portal. The Web-server acts as a mediator sending the users' requests to the database nodes and initiating their distributed evaluation. Each request is broken down into multiple parts based on the spatial layout of the data. Each part is asynchronously submitted for evaluation to the database which stores the data needed for the evaluation. The Web-server assembles the results from the distributed computation and sends them back to the client.

The JHTDB provides a variety of data-intensive analysis routines that are executed on the database nodes. These include interpolation, differentiation, particle tracking and spatial filtering. These tasks are often data-intensive and in order to leverage the capabilities of the cluster we have developed data-driven batch processing techniques for their evaluation [17, 16]. Most of these tasks usually operate on subsets of the space or a collection of individual target locations within a time-step.

In contrast, threshold and top-k queries usually have to examine the entire data volume of a time-step or a signification portion of it. Furthermore, the data product of threshold queries is much smaller in relation to the amount of data that need to be examined. This fact combined with the fact that subsequent queries can reuse previously computed results makes the query results suitable to caching.

3. SCIENTIFIC USE CASES

One of the applications of thresholding in turbulence is to find the locations of maximum vorticity in a particular time-step or the dataset as a whole. The locations of maximum vorticity are usually associated with the most intense vortices in the dataset and often have interesting and complex reconnection events associated with them. Once obtained from the service, these locations can be clustered in both 3d and 4d. This allows scientists to examine their evolution with the flow and make subsequent analysis queries as needed in order to study these events. The relationship between different "worms" (see Figure 3) that connect and reconnect at those locations is of the most interest.

The vorticity is computed from the velocity field by taking its curl:

$$\begin{aligned}\vec{\omega} = \vec{\nabla} \times \vec{v} &= \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}\right) \times (v_x, v_y, v_z) \\ &= \left(\frac{\partial v_z}{\partial y} - \frac{\partial v_y}{\partial z}, \frac{\partial v_x}{\partial z} - \frac{\partial v_z}{\partial x}, \frac{\partial v_y}{\partial x} - \frac{\partial v_x}{\partial y}\right). \qquad (1)\end{aligned}$$

We use finite differencing methods of different orders for the evaluation of the curl. For example, with 4th-order centered

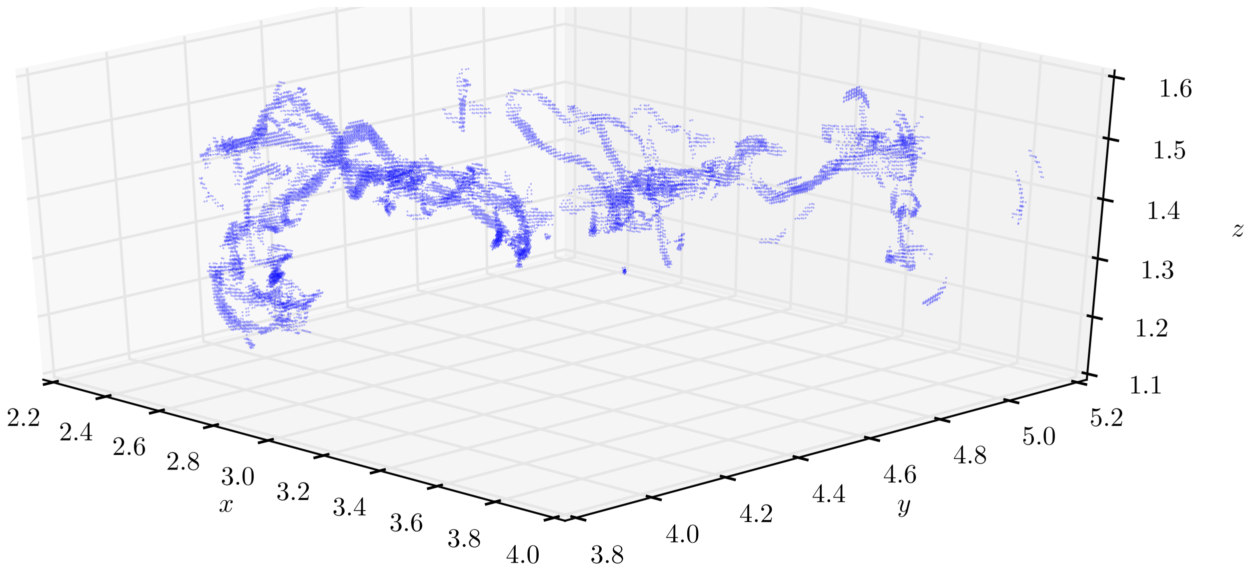


Figure 3: 3D (single time-step) cut through the 4D cluster containing the most intense event.

finite differencing each partial derivative is evaluated from the 4 adjacent grid node values as follows:

$$\left.\frac{df}{dx}\right|_{x_n} = \frac{2}{3\Delta x}[f(x_{n+1}) - f(x_{n-1})] - \frac{1}{12\Delta x}[f(x_{n+2}) - f(x_{n-2})], \qquad (2)$$

where f denotes any one of the three components of the velocity and Δx is the width of the grid in the x direction. The partial derivatives along y and z are computed in the same fashion. Figure 2 shows the distribution of the values of the norm of the vorticity field in the MHD dataset for a representative time-step. This is indicative of how the values are distributed in the dataset as a whole. This coarse view of the data can be used by scientists to guide the selection of threshold values.

Figure 3 shows the most intense event observed in the forced isotropic turbulence dataset. The locations of maximum vorticity in the dataset were clustered in this case in 4d using a friends-of-friends algorithm. It is interesting to note that the cluster containing the most intense event in the entire dataset develops from nothing (i.e. it does not appear in the first few time-steps) and it takes less than the timespan stored in the database for it to develop. Figure 3 also shows that most interactions between worms are not simple. There are several worms interacting in a complex way at the same time. Similar type of analysis and the fact that the entire time history of the simulation is available in a database cluster, which provides built-in sophisticated analysis routines revealed flux-freezing breakdown in MHD turbulence [12], showing why solar flares last minutes rather than the millions of years that conventional theory would predict.

In addition to obtaining the regions of largest vorticity, there is substantial interest in studying the regions with highest values for other fields, such as the second and third velocity gradient invariants (Q and R). These invariants are scalar quantities whose values contain information about the topology of the flow and the rates of vortex stretching and rotation. In MHD, finding the locations with largest values for the electric current can lead to new insights into the development of the most intense reconnection events of magnetic field sheets in the simulated plasma. Similarly to the vorticity, the electric current is derived from the magnetic field by taking its curl. The list of fields of interest, on which scientists would like to perform threshold queries certainly does not stop here and is indicative of how valuable this functionality is in the study of turbulence and fluid dynamics.

4. THRESHOLD QUERY EVALUATION

Threshold queries of derived fields submitted to the JHTDB are evaluated using a data-parallel execution strategy and the query results are cached in an application-aware semantic cache. In addition to query results, the cache stores their semantic descriptions and query metadata and parameters used to obtain them. The evaluation strategy for queries that do not hit the cache is driven by the spatial partitioning of the data across the nodes of the cluster.

Derived fields computation: The databases store only the raw field data from the simulation (e.g. velocity, pressure, magnetic field etc.). However, the threshold queries of most interest to science users produce all locations where the values of a *derived* field are above a given threshold. Thus, the derived field in question has to be computed from the raw data first. For most derived fields of interest, this computation has local support. It has an associated localized kernel of computation around each grid node. Therefore, the value of the derived field at each grid node depends on the value of the stored field at all of the grid locations, which are part of the kernel of computation.

Distributed data-parallel execution: In most cases threshold queries operate over an entire time-step. Each such query is subdivided by the mediator into queries submitted to each of the database nodes. Each node evaluates the query over the data that it has stored locally. Only a

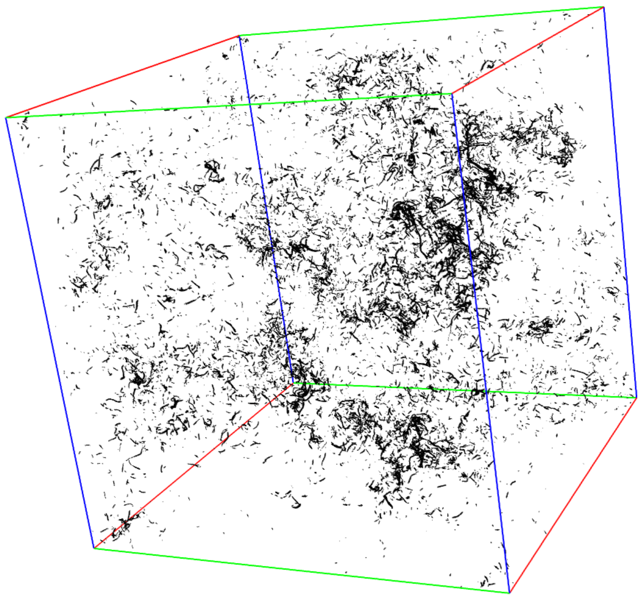


Figure 4: Points with values above 7 times the root mean square value of the vorticity for a single time-step.

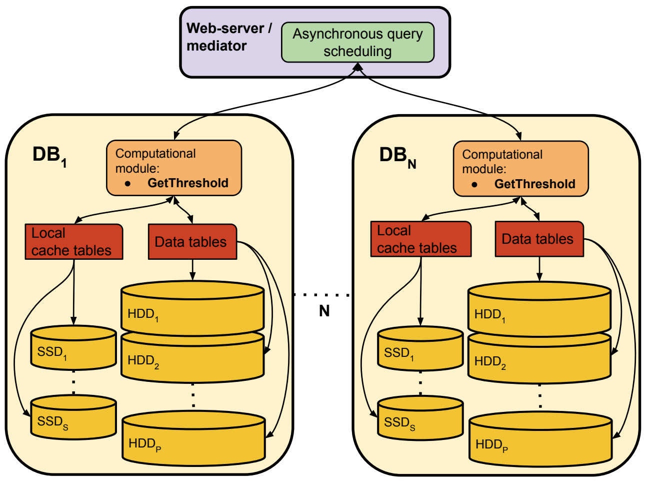


Figure 5: Distributed evaluation of threshold queries and architecture of the application-aware cache. Each database node has local cache tables, which reside on solid-state drives attached to the node.

small amount of data along the boundary need to be requested from adjacent nodes. The size of the band of data that may not be available locally is equal to a kernel half-width. Such a band is needed on each of the sides of the box forming the domain of the computation. The data are read into memory and the particular field requested is computed at each of the locations on the grid. The same strategy applies when utilizing multiple processes per node. The norm or absolute value of the derived field at each location is compared against the specified threshold and if the value is higher, it is maintained along with the spatial coordinates of the location in a list.

We impose a limit on the maximum number of locations that can be returned as a result of a threshold query in order to prevent having to return the entire time-step or a significant fraction of it for queries with thresholds that are set too low. Currently this limit is set conservatively to 10^6 locations, which is sufficient to examine a time-step in detail. In the case of the vorticity, the values above 8 times the root mean square value, which is about 25% of the maximum, are contained within 2.6×10^5 points in each time-step. Figure 4 shows all the points in a single time-step with values above 7 times the root mean square value. There are 2.4×10^5 points in the figure. Given that we are interested in extreme events, obtaining the locations with values even within 50% of the maximum would be sufficient. At the same time this also limits the amount of data that have to be returned to the user over the network as well as the amount of data that have to be cached. Users receive an error message notifying them if their request has a threshold that is set too low. If a user is interested in obtaining more data he or she can request the values of the derived field directly. Alternatively, if they are interested in the density distribution of values they can examine the probability density function (e.g. Fig. 2), which is computed using a similar strategy to threshold queries.

Application-aware cache for query results: A central part of the evaluation strategy for threshold queries that we have developed is the introduction of an application-aware cache for query results (Fig. 5). The results of these queries are small compared to the amount of data that need to be examined and the results can be used to answer subsequent queries as long as they are within the same region and specify the same or higher threshold. Each database node has a local cache. Cache entries are indexed by the field, time-step, spatial region and the threshold requested. We use a least recently used cache replacement policy. All modifications of and queries to the cache are executed within a transaction with snapshot isolation level to avoid dirty-reads or an inconsistent view of the cache.

Caching the query results preserves the computational effort in addition to substantially reducing I/O. The cached data are for the particular derived field that was queried and not the raw data of the simulation fields. Thus, we do not have to derive the requested field from the raw data for queries that hit the cache. This results in a substantial improvement in query performance as we only have to scan a small set of data and do not need to perform any additional computation.

Not all query results are suitable to caching. Most of the queries submitted to the JHTDB other than threshold queries request data at a collection of target locations. Given that there are 1024^4 possible locations for three of the datasets and $6 * 1024^4$ locations for the channel-flow dataset the chance of reuse for the results of these queries is extremely small. This is why the cache currently stores only the results of threshold queries. Nevertheless, it can easily be extended to cache the results of other query types as well if that becomes advantageous.

The cached query results are stored in a table in the database and the overall size of the cache is limited by the amount of available SSD disk space, not memory. Given a limit of $\sim 10^6$ points per time-step for a threshold query, the space required to cache the maximum number of points including the index space and database overhead is ∼40MB. Therefore for a dataset containing 1024 time-steps, as is the case for the isotropic turbulence and MHD datasets part of the JHTDB, a cache size of 40GB is sufficient to cache the query results for threshold queries of a derived field over the entire dataset. The currently available SSD disk space per node is ∼200GB, which will be sufficient to maintain the threshold results for nearly five derived fields over the entire dataset. In contrast, computing and materializing a scalar derived field for the entire dataset would require ∼5TB (15TB for vector fields).

Algorithm 1 Get points above threshold using cache

```
Require: Dataset $d$, Field $f$, Timestep $t$, Threshold $k$,
    Query box $q = [x_l, y_l, z_l, x_u, y_u, z_u]$
 1: procedure GetThreshold
 2:     $points \leftarrow List()$
 3:     $updateCache \leftarrow false$
 4:     $query \leftarrow$ SELECT * FROM cachedb..cacheInfo
                WHERE dataset = $d$ AND field = $f$
                AND timestep = $t$
 5:     $command \leftarrow SqlCommand(query)$
 6:     $reader \leftarrow command.ExecuteReader()$
 7:     if $reader.HasRows()$ then
 8:         $k_s \leftarrow reader[\text{"threshold"}]$        ▷ Stored threshold
 9:         $start \leftarrow reader[\text{"startIndex"}]$
10:         $end \leftarrow reader[\text{"endIndex"}]$
11:         $ordinal \leftarrow reader[\text{"ordinal"}]$
12:         if $k \geq k_s$ & $q \in$ [start, end] then
13:             $query \leftarrow$ SELECT * FROM cachedb..cacheData
                    WHERE cacheInfoOrdinal = $ordinal$
14:             $command \leftarrow SqlCommand(query)$
15:             $reader \leftarrow command.ExecuteReader()$
16:             while reader.Read() do
17:                 $location \leftarrow reader[\text{"zindex"}]$
18:                 $norm \leftarrow reader[\text{"dataValue"}]$
19:                 if $norm \geq k$ & $location \in q$ then
20:                     $points.Add(new\ Point(location, norm))$
21:                 end if
22:             end while
23:         else
24:             $updateCache \leftarrow true$
25:         end if
26:     else
27:         $updateCache \leftarrow true$
28:     end if
29:     if $updateCache$ then
30:         Retrieve data covering $q$ from DB.
31:         for all $p \in q$ do
32:             Compute $f$ at $p$.
33:             if $\|f(p)\| \geq k$ then
34:                 $points.Add(new\ Point(p, \|f(p)\|))$
35:             end if
36:         end for
37:         Update cacheInfo and cacheData tables.
38:     end if
39:     return $points$
40: end procedure
```

The entire cache is comprised of two database tables. The *cacheInfo* table stores metadata for the cached entries. It stores information about the dataset, field, time-step, start and end coordinates of the spatial region examined and the threshold value used. The *cacheData* table stores the locations of all of the grid points, for which the field queried has a norm higher than the specified threshold. The *cacheData* table is foreign key constrained with the ordinal of the *cacheInfo* table. This allows us to quickly find a record in the *cacheInfo* table and retrieve all of the cached entries using an index lookup.

Overall execution of threshold queries: Algorithm 1 illustrates the process of obtaining all points with norms of the specified field above the given threshold from the database in the presence of a cache. The mediator submits a query to each of the database nodes storing the raw data asynchronously. Each node begins evaluation of the query by executing Algorithm 1. First a cache lookup is performed. If the data for the requested field, time-step and spatial region are available in the cache and if the specified threshold is higher than the one stored in the cache the query can be answered from there. The records are retrieved from the cache and the ones that have a higher value are returned to the mediator and subsequently back to the user. If the data stored in the cache have a higher threshold than the one requested the cache needs to be updated. Similarly, if the cache does not have an entry for the specified parameters the query needs to be evaluated from the raw data. In those cases the raw data are read into memory with data along the boundary requested from adjacent nodes as needed. The specified field is derived at each location on the grid and the norm or absolute value of the field is compared against the threshold. The locations where the values are higher than the threshold need to then be stored in the cache. If the cache does not have enough space for the new records, space is freed up by removing the least recently used data across all quantities. Reading from, updating or modifying the cache is done within a transaction with snapshot isolation level. Snapshot isolation allows us to avoid locking the tables that serve as the cache for each transaction. This provides for a higher degree of parallelism and avoids any potential deadlocks from queries running in parallel.

5. EXPERIMENTAL RESULTS

We evaluate the developed method for the execution of threshold queries to large numerical simulation datasets with the goal of analyzing the benefits and overhead from the introduction of the application-aware cache. We also analyze the scaling properties of the method. Finally, we show that an integrated method that performs the evaluation on the database nodes near the data is several orders of magnitude faster than the user requesting the derived filed of interest from the database and evaluating the threshold locally.

5.1 Experimental Setup

The experiments were run on the production database nodes of the JHTDB through a development Web-server hosting the Web-services. We used the MHD dataset (Sec. 2) for the experimental runs. This dataset is partitioned across 4 database nodes according to spatial regions in the Morton z-order. The database nodes are 2.66 GHz dual quad-core Windows 2008 servers with SQL Server 2008 R2 and 24 GB of memory. Each node has 24 2TB SATA disks arranged as a set of four RAID-5 arrays. The database files are striped across the nodes and their associated disk arrays. The tables storing the data are partitioned spatially along

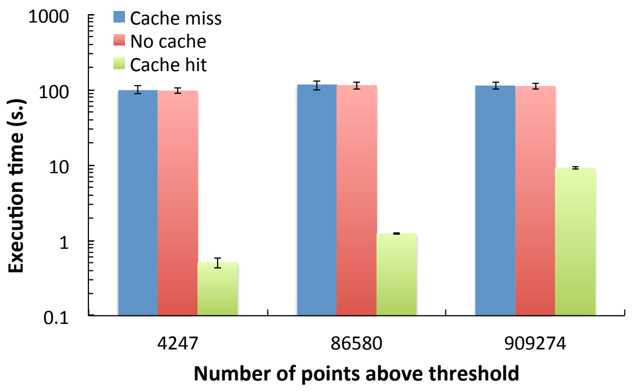


Figure 6: Execution time for threshold queries at different threshold levels compared with the execution time of the same queries in the absence of a cache.

contiguous ranges of the Morton z-curve and the data for each partition reside in one database file.

For this evaluation we looked at the performance of threshold queries to the vorticity field. The vorticity field is representative of derived fields that have to be computed from the stored data. It is defined as the curl of the velocity field. As described in section 3 thresholding the vorticity field is important in the study of fluid dynamics and obtaining the locations of maximum vorticity can lead to new insights into the development of the most intense vortices observed in the dataset.

5.2 Evaluation of cache effectiveness

The central part of the strategy that we have developed for the evaluation of threshold queries of derived fields is the application-aware cache, which stores the results of these queries. We first evaluate the overhead associated with the introduction and maintenance of the cache. Figure 6 compares the execution time of queries in the absence of a cache with the execution time of the same queries, which interrogate the cache first (blue and red bars in the figure). The execution times are also shown in Table 1. For these experiments we requested the locations with norms of the vorticity above thresholds at different levels. We refer the reader to Figure 2, which shows the distribution of values of the norm of the vorticity field in the MHD dataset to get an appreciation of the different threshold values used in the experiments. For the first set the threshold was set high (80.0) and only ∼4,300 points (or 0.0004% of all points) were above the threshold. For the second set a medium threshold (60.0) was chosen and ∼87,000 points (or 0.0081% of all points) were above the threshold. Finally, a low threshold (44.0) was chosen for the last set and there were ∼900,000 points (or 0.0847% of all points) above the threshold. For each set a random time-step was chosen and the queries were run against that time-step. The measurements were taken from the point of view of the end user.

As we can see from the results shown in Figure 6 the overhead associated with querying the cache first is minimal, less than 3% and within the margin of error. The cache was initially populated by executing several hundred unrelated queries and contained several million entries. During the "cache-miss" runs cache entries for the particular time-step queried were dropped before each run, making sure that each query would produce a cache miss and would have to be evaluated from the raw data. The execution times were averaged over 10 runs. We utilized 4 processes per database node for the evaluation of each query. The method shows stable running time across different time-steps and threshold levels in the absence of a cache and during cache misses. The running time increases slightly only because of the larger result set that has to be returned to the user.

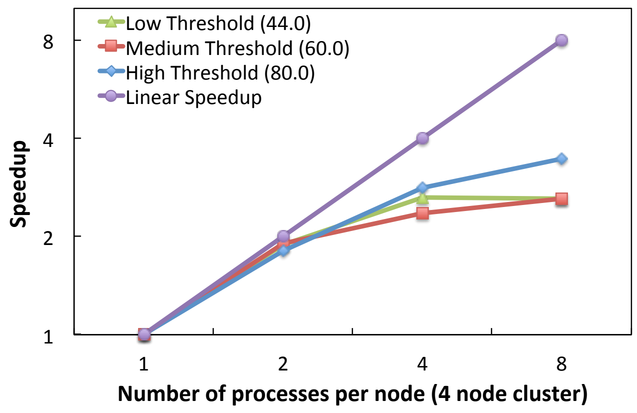
Vorticity threshold	Points above threshold	Average Running time (s.)		
		No cache	With cache (miss)	With cache (hit)
80.0	4247	97.1	100.2	0.5
60.0	86580	113.7	115.9	1.2
44.0	909274	111.6	115.0	9.1

Table 1: Effectiveness of caching.

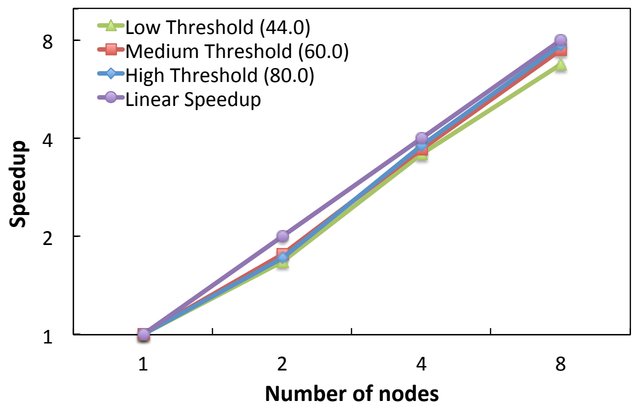
Cache hits reduce the running time of threshold queries by over an order of magnitude as shown in Figure 6 and Table 1. This is because we do not have to compute the requested derived field from the raw data, which eliminates the associated I/O. Only the cache entries need to be looked-up, which is substantially less data than the raw vector or scalar field data. For the queries with large result sets it is actually the network time taken to transfer the results to the user that dominates the overall execution as opposed to the I/O or computation time as we show later. Cache hits are evaluated by first warming up the cache by submitting the same set of threshold queries of the vorticity field as before. We then submit several more unrelated queries with different time-steps and threshold values in order to pollute the cache. Finally, we issue the original set of queries and measure their running times. Let us focus on the query with low threshold, which returns ∼900,000 points. Given that valid threshold values are limited to those that result in no more than 1,000,000 points it is likely that all subsequent queries to this time-step will result in a cache hit as their threshold is likely to be equal or higher than the cached one. Currently we observe fairly high cache-hit ratios as the workload is very structured and queries tend to examine the same regions in space and time.

5.3 Scaling and Distributed Evaluation

The evaluation of threshold queries of derived fields from the raw data is both I/O and computationally bound. These queries examine the entire data volume of a simulation time-step and are, therefore, good candidates for a data-parallel distributed evaluation. Our data-parallel implementation exhibits good vertical and nearly ideal horizontal scaling as shown in Figure 7. For the scale-up experiments (Fig. 7(a)), we used the same queries and threshold values as for the runs shown in Figure 6 and Table 1 but with varying number of processes per node. Cache entries for the time-step queried were again dropped before each run in order to evaluate the scaling properties of the computation of the derived field from the stored data. The computations for all of the derived fields of interest (such as the vorticity) at each grid point need data from adjacent grid points only. Therefore, each node of the cluster is able to compute the derived field



(a) Scale-up with multiple processes per node



(b) Scale-out to multiple nodes

Figure 7: Execution time for threshold queries at different threshold levels – high, medium and low. The scale-up evaluation was performed utilizing 1-8 processes per server on a 4-node cluster. The scale-out evaluation was performed on 1 through 8 nodes.

from data available locally with only a small amount along the boundary of each region having to be retrieved from adjacent nodes. Each computation is independent and embarrassingly parallel. This allows us to make use of multiple processes per node and scale out to multiple database nodes.

We observe nearly a two times speedup when going from a single process per node to two processes per node (Fig. 7(a)). The speedup diminishes to 1.4 times when going to 4 processes and little speedup is observed with 8 processes per node. While the computation time scales with increased process count, the time to perform I/O does not as the data on each node reside in the same database table and on the same set of disks. Additionally, I/O redundancy increases as the process count increases as data along the boundary of each region are requested by multiple processes. SQL Server already utilizes parallelism to perform the I/O even when data are retrieved utilizing a single query. Finally, the experiments were run on the live production database nodes, which were also servicing other user queries in addition to operating system and other SQL Server processes. Nevertheless, running with 4 processes per node is nearly 2.6 times faster when compared to running with a single process.

The scale-out experiments show a nearly perfect linear speedup as the evaluation is distributed to an increasing number of database nodes (Fig. 7(b)). For these experiments we issued queries with the same threshold levels as before to a cold cache. We utilized a single process per database node to evaluate the horizontal scaling of the computation. The evaluation benefits not only from the additional computational resources with the addition of database nodes to the cluster but also from the increased memory size. The data needed for the computation of each derived field are read into memory and the larger memory size means that there is less contention with other system and application processes and it is less likely that virtual memory needs to be used. SQL Server also benefits from a larger buffer pool, which reduces the I/O time.

As expected, we observe even weaker speedup when the queries perform nothing but I/O and the number of processes per node is increased. Figure 8 compares the running time of the queries with a medium threshold and executed

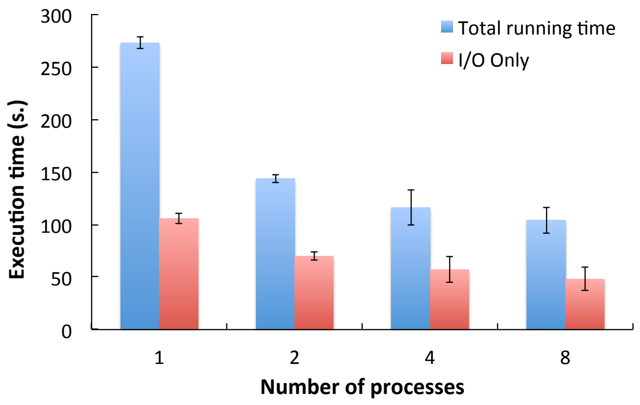


Figure 8: Execution time for threshold queries evaluated utilizing different number of processes per server compared with the time taken to perform the I/O only.

with varying number of processes per node with the time taken to perform the I/O only. The I/O time is about half of the total running time for these queries. SQL Server already makes use of parallelism internally and the data have to be retrieved from the same set of disks. Nevertheless, the I/O time does decrease with additional processes, this is because the data reside in a partitioned table and the data in each partition are placed in a separate file on one of the disk arrays. Depending on how the data requests are scheduled in SQL Server this allows for the disks arrays to be driven in parallel. Additionally, with more processes per node the data can also be consumed faster. It is worth noting that the total running time for the queries evaluated with 4 or 8 processes is about the same as the time it takes to perform the I/O only with a single process.

So far we have presented the effectiveness of evaluating threshold queries of derived fields on the database cluster storing the raw simulation data. The data-parallel computation of the derived fields allows us to evaluate a threshold

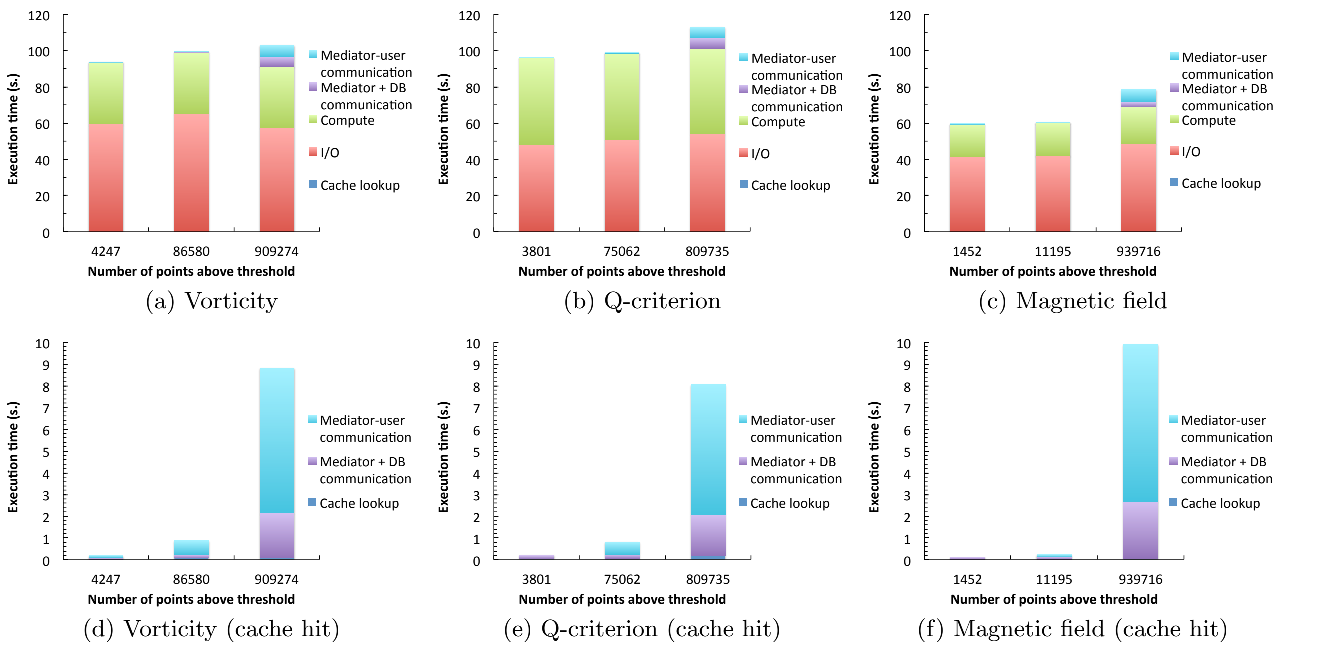


Figure 9: Breakdown of the execution time for threshold queries requesting different fields and at different threshold levels – high, medium and low.

query over an entire 1024^3 time-step part of a 20TB dataset in less than two minutes. The introduction of an application-aware cache for the query results of these queries reduces this time to several seconds when there is a cache hit. In contrast, one of our science collaborators reported that his evaluation of this functionality performed locally would take over 20 hours to complete. To perform the evaluation locally the user requests the derived field of interest from the database by submitting multiple queries over subregions of a time-step. This is necessary as requesting a derived field over an entire time-step will overload the network. Derived fields may have even more components than the scalar or vector field stored in the database. For example, the velocity gradient (needed for the computation of the vorticity) has 9 components compared with the 3 components of the velocity. Given a single-precision floating-point representation, this makes the velocity gradient of an entire time-step at least 36GB in size. A Web-service request will be much larger due to the overhead of wrapping the data in an xml format. After the field of interest is obtained locally the user has to threshold it to get the final result, which is reasonably fast, but discards most of the data that have been requested to yield a small in size result.

5.4 Evaluation of Additional Fields

The data-parallel evaluation of threshold queries shows stable execution time for different derived fields in addition to the different threshold levels and time-steps queried. The execution times depend on the complexity of the computation needed to evaluate the particular derived field requested. Figures 9(a), 9(b) and 9(c) show a breakdown of the execution time of threshold queries of different derived fields, which are evaluated from the raw data and a cold cache using 4 processes per node on a 4 node cluster. Almost the entire time is spent performing the I/O and computation associated with the derived field requested.

The vorticity field and the Q-criterion have similar I/O requirements as they have the same kernels of computation and are both derived from the velocity gradient. The vorticity has 3 components and its computation only examines 6 of the 9 components of the velocity gradient, which are also examined in pairs (see Eq. 1). On the other hand, even though the Q-criterion is a scalar value, it is computed through a non-linear combination of all 9 of the components of the velocity gradient. This means that the velocity gradient has to be computed at each grid location before the Q-criterion is evaluated, which is reflected in the increased computation time that we observe for the Q-criterion. The magnetic field is one of the raw fields of the magnetohydrodynamics dataset that are stored in the database. Therefore, there is no additional computation needed to derive it from the data, every data point has to be simply compared with the threshold level specified. This is why the computation time is much smaller compared to the queries for the vorticity and the Q-criterion. The I/O time for the magnetic field is also smaller. This is because its kernel of computation is a single point and therefore there are no additional data along the boundary that have to be requested from adjacent nodes. In that case all of the data needed are available locally for each database node.

In all of these cases, the time taken to interrogate the cache is negligible. The mediator time to distribute the queries and assemble the results as well as the time to transfer them to the user are also substantially smaller than the I/O and computation times. As expected they increase proportionally to the number of points in the result set.

It is interesting to note that the time taken to perform a cache lookup is relatively small even in the case of a cache hit as can be seen in Figures 9(d), 9(e) and 9(f). This is because the cache tables reside on SSDs attached to each database node (see Fig. 5) and retrieving the data is always done through a clustered index lookup. In the cases of a cache hit, the majority of the time is spent simply transferring the results from the database nodes to the mediator and then back to the user. These times remain more or less constant between the cases of cache misses and cache hits (top row and bottom row of Fig. 9). Caching the results of threshold queries effectively preserves the I/O and computational effort spent during their initial evaluation and results in over an order of magnitude speedup for all the different fields requested as we can see in Figure 9.

6. RELATED WORK

Only select few database systems offer support for arrays as first-class citizens. Even fewer provide the fault-tolerance, scalability and availability guarantees necessary for a system managing multi-terabyte datasets in a production setting. This is part of the reason why we have chosen to represent the array data in the JHTDB as a collection of binary large objects in a relational DBMS and perform the array manipulation tasks necessary at the application level. The systems that provide support for arrays and aim to handle large array data efficiently are RasDaMan [5], SciDB [30] and MonetDB/SciQL [34]. RasDaMan partitions raster objects into tiles, which are stored in a traditional relational database system. This approach is similar to how the numerical simulation data are handled in the JHTDB. RasDaMan provides RasQL [4], which is a SQL-92 based query language for the manipulation of raster images. SciDB is an array database system build from the ground up. Array attributes are partitioned vertically and each attribute array is decomposed into overlapping chunks. SciDB provides a declarative Array Query Language (AQL) and an Array Functional Language (AFL). Users can create arrays with named dimensions with AQL and make use of the functional operators defined in AFL, such as SLICE, SUBSAMPLE, SJOIN, FILTER and APPLY. SciQL's focus is on language design and integration with SQL:2003 syntax and semantics. It is implemented within the MonetDB framework [24].

Database systems support rollup queries, including top-k queries, but in most cases these queries apply only simple linear score functions on the attribute values of individual records. Additionally, many top-k query evaluation techniques rely on the score functions being monotone in order to perform early pruning (see [15] for a survey of top-k evaluation strategies). This is an assumption that we cannot make for the functions used to compute all the different possible derived fields of interest in fluid mechanics. Even approaches that aim to work with general score functions [11, 33] assume that the function operates on the attributes of a single record. In contrast, our approach performs a kernel computation at each grid location in order to obtain the value of a *derived* field at that location and examines the vector or scalar array data at all neighboring locations, which are within the kernel of the computation. The functions used to derive the field may even be non-linear. Finally, a top-k approach may not be suitable in the cases where scientists are interested in performing threshold queries at different time-steps as the same threshold level will produce different number of points in the result set for different time-steps.

The processing of top-k queries has been studied extensively in the context of distributed and relational database systems. A survey of different techniques in the case of centralized processing is given in [15]. In the case of distributed processing different approaches focus on horizontally [3, 32] or vertically [7, 8, 13, 21, 22] distributed data. None of these approaches deal with array data stored in a relational database system. Zhao et al. propose an algorithm for the processing of top-k queries in large-scale distributed environments called BRANCA [35]. They build on the idea of semantic caching [27] and make use of branch caches, which store results of previous top-k queries with respect to the data stored on each server. The caching mechanism that we use is similar in that regard, but the queries that are evaluated in our system operate on derived fields, which are computed at each location by accessing data from a surrounding region. The queries described in [35] operate over the attributes of individual records only using simple linear score functions.

Aßfalg et al. introduce the concept of threshold queries in time-series databases [2]. Their definition of threshold queries differs from the threshold queries described in this paper. They are concerned with determining the time-series, which exceed a user-defined threshold at time frames similar to the time-series specified in the query. Thus, their definition of threshold queries is concerned with the temporal relationship between the time-series stored in the database (usually one dimensional sequence of measurements) and the time-series given in the query. In contrast, our approach focuses on reporting all of the spatial locations of a multidimensional field where the norm or absolute value of the field exceeds a user prescribed threshold.

In a system called the tree cache, Lopez et al. [20] make use of a small application-aware cache to reduce access time to large datasets stored on disk. The tree cache stores individual octants of octree datasets and exploits application-specific information to determine which octants to cache and to perform query reordering. This work has inspired the use of an application-aware cache for the evaluation of threshold queries. In contrast to the tree cache, we do not cache raw data objects, but rather query results. Caching query results preserves the computational effort in addition to reducing I/O, which has a much bigger impact on query performance and substantially reduces the size of the cache. Additionally, the cache that we introduce resides on disk rather than in memory, which greatly increases its potential size. Lopez et al. also explore approximate querying through aggregation, which can be fairly easily supported by our system but is of limited use as scientists performing threshold queries are usually interested in obtaining the exact locations where a field is at its highest values.

Sampling approaches [29, 25] offer an alternative to the on-demand computation of derived fields and the evaluation of threshold queries on them. The goal of both techniques is to not return large data volumes, but focus on the most intense events and interesting regions in the dataset. The computation of derived fields is carried out on the nodes of the database cluster and takes a look at the dataset as a whole, while the user obtains only a small subset of the data, where the derived field in question is above the prescribed threshold. Sampling approaches can potentially omit some locations and while useful for generating initial impressions may not be suitable if the exact locations where a field is at its highest values are desired.

Andrade et al. [1] describe a database system and an optimization framework build on the concept of *active se-*

mantic caching. An active semantic cache aims to fully or partially reuse cached query results or aggregates through automated transformations of these aggregates. Similarly to our work they focus on real scientific data-analysis applications. The method that we have developed for the evaluation of threshold queries complements the active semantic caching approach and could be used in that framework. Our work has focused on extending a relational database system (Microsoft's SQL Server) as opposed to designing a new database system from the ground up as described by Andrade et al. [1].

7. CONCLUSIONS AND FUTURE WORK

We have presented an efficient strategy for the evaluation of threshold queries of derived fields in large numerical simulation datasets. The thresholded fields are derived from the stored simulation data in a distributed data-parallel manner. The computations scale with the cluster resources and are performed on the database nodes, where the data are stored. This new capability allows researches to quickly obtain and focus on regions of special interest even if they lack the computing capabilities or data transfer rates necessary to examine entire time-steps or large parts of the entire dataset.

We have introduced an application-aware cache for the query results of threshold queries. Cache hits reduce query running times by over an order of magnitude. The cache adds minimal overhead during the evaluation of queries even if there is a cache miss and has modest storage requirements. The cache is represented as a set of database tables and resides on disk rather than in memory. Each database node has local cache tables, which allows the cache to scale-out as the cluster grows.

The introduction of an application-aware cache for query results lays the groundwork for the creation of a landmark database. Such a database can store the locations of the highest vorticity regions in the dataset or more broadly regions of interest and their associated statistics.

The Web-services approach to archived numerical simulation datasets provides public access to high quality simulation data to anyone with an internet connection. The Web-services methods can be called from any modern programming language and we provide C, Fortran and Matlab client libraries for the JHTDB. The evaluation of each query submitted through a Web-service call is carried out on the nodes of the database cluster by means of a stored procedure or a user-defined function that has been implemented and deployed to handle these requests. This allows us to fine-tune the execution of these procedures and handle all requests transparently to the user. However, this approach also has drawbacks. Adding new functionality means adding to a long list of Web-service calls and requires substantial implementation effort. In the case of threshold queries the stored procedure performing the evaluation must have an implementation for each derived field of interest even though the execution is handled by the same Web-service call.

In the future, we plan to develop declarative and graphical user interfaces that will allow users to combine existing building blocks and perform computations that have not been explicitly implemented. Additionally, we plan on deploying a server-side computing environment for users similar to the CasJobs service for the Sloan Digital Sky Survery [18]. In such an environment users can run queries in batch mode and save their results in a personal database called MyDB, which resides on the servers near the data. This will allow for much greater flexibility in the type of computations that can be performed in addition to substantially decreasing the network overhead.

8. ACKNOWLEDGMENTS

The authors would like to thank the Turbulence Database Group at Johns Hopkins University as well as three anonymous reviewers for their insightful comments and suggestions. This work is supported in part by the National Science Foundation under Grants CMMI-0941530, ACI-1261715, OCI-1244820 and AST-0939767 and Johns Hopkins University's Institute for Data Intensive Engineering & Science.

9. REFERENCES

[1] H. Andrade, T. Kurc, A. Sussman, and J. Saltz. Active semantic caching to optimize multidimensional data analysis in parallel and distributed environments. *Parallel Computing*, 33(7-8):497–520, Aug. 2007.

[2] J. Aßfalg, H.-P. Kriegel, P. Kröger, P. Kunath, A. Pryakhin, and M. Renz. Similarity Search on Time Series based on Threshold Queries. In *EDBT*, 2006.

[3] W.-T. Balke, W. Nejdl, W. Siberski, and U. Thaden. Progressive Distributed Top-k Retrieval in Peer-to-Peer Networks. In *ICDE*, 2005.

[4] P. Baumann. A Database Array Algebra for Spatio-Temporal Data and Beyond. In *Proceedings of the 4th International Workshop on Next Generation Information Technologies and Systems*, NGIT '99, 1999.

[5] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The Multidimensional Database System RasDaMan. In *SIGMOD*, 1998.

[6] R. Burns, K. Lillaney, D. R. Berger, L. Grosenick, K. Deisseroth, R. C. Reid, W. G. Roncal, P. Manavalan, D. D. Bock, N. Kasthuri, M. Kazhdan, S. J. Smith, D. Kleissas, E. Perlman, K. Chung, N. C. Weiler, J. Lichtman, A. S. Szalay, J. T. Vogelstein, and R. J. Vogelstein. The open connectome project data cluster: Scalable analysis and vision for high-throughput neuroscience. In *SSDBM*, 2013.

[7] P. Cao and Z. Wang. Efficient top-K Query Calculation in Distributed Networks. In *PODC*, 2004.

[8] S. Chaudhuri, L. Gravano, and A. Marian. Optimizing Top-k Selection Queries over Multimedia Repositories. *IEEE Trans. on Knowl. and Data Eng.*, 16(8):992–1009, Aug. 2004.

[9] S. Dar, M. J. Franklin, B. T. Jónsson, D. Srivastava, and M. Tan. Semantic Data Caching and Replacement. In *VLDB*, 1996.

[10] DataScope. http://idies.jhu.edu/datascope.

[11] P. M. Deshpande, D. P, and K. Kummamuru. Efficient Online top-K Retrieval with Arbitrary Similarity Measures. In *EDBT*, 2008.

[12] G. Eyink, E. Vishniac, C. Lalescu, H. Aluie, K. Kanov, K. Bürger, R. Burns, C. Meneveau, and A. Szalay. Flux-freezing breakdown in high-conductivity magnetohydrodynamic turbulence. *Nature*, 497(7450):466–9, 2013.

[13] U. Güntzer, W.-T. Balke, and W. Kießling. Optimizing Multi-Feature Queries for Image Databases. In *VLDB*, 2000.

[14] A. J. G. Hey, S. Tansley, and K. M. Tolle. *The Fourth Paradigm: Data-Intensive Scientific Discovery.* Microsoft Research, 2009.

[15] I. F. Ilyas, G. Beskales, and M. A. Soliman. A Survey of Top-k Query Processing Techniques in Relational Database Systems. *ACM Comput. Surv.*, 40(4):11:1–11:58, Oct. 2008.

[16] K. Kanov, R. Burns, G. Eyink, C. Meneveau, and A. Szalay. Data-intensive Spatial Filtering in Large Numerical Simulation Datasets. In *Supercomputing*, 2012.

[17] K. Kanov, E. Perlman, R. Burns, Y. Ahmad, and A. Szalay. I/O Streaming Evaluation of Batch Queries for Data-intensive Computational Turbulence. In *Supercomputing*, 2011.

[18] N. Li and A. R. Thakar. CasJobs and MyDB: A Batch Query Workbench. *Computing in Science and Engineering*, 10(1):18–29, 2008.

[19] Y. Li, E. Perlman, M. Wan, Y. Yang, C. Meneveau, R. Burns, S. Chen, A. Szalay, and G. Eyink. A public turbulence database cluster and applications to study Lagrangian evolution of velocity increments in turbulence. *Journal of Turbulence*, page N31, 2008.

[20] J. C. Lopez, D. R. O'Hallaron, and T. Tu. Big Wins With Small Application-aware Caches . In *Supercomputing*, 2004.

[21] A. Marian, N. Bruno, and L. Gravano. Evaluating Top-k Queries over Web-accessible Databases. *ACM Trans. Database Syst.*, 29(2):319–362, June 2004.

[22] S. Michel, P. Triantafillou, and G. Weikum. KLEE: A Framework for Distributed Top-k Query Algorithms. In *VLDB*, 2005.

[23] The Millennium Simulation. http://www.mpa-garching.mpg.de/millennium/.

[24] MonetDB. http://monetdb.cwi.nl/.

[25] S. Nirkhiwale, A. Dobra, and C. Jermaine. A sampling algebra for aggregate estimation. *Proc. VLDB Endow.*, 6(14):1798–1809, Sept. 2013.

[26] E. Perlman, R. Burns, Y. Li, and C. Meneveau. Data Exploration of Turbulence Simulations Using a Database Cluster. In *Supercomputing*, 2007.

[27] Q. Ren, M. H. Dunham, and V. Kumar. Semantic Caching and Query Processing. *IEEE Trans. on Knowl. and Data Eng.*, 15(1):192–210, Jan. 2003.

[28] The Sloan Digital Sky Survey. http://www.sdss.org/.

[29] L. Sidirourgos, M. L. Kersten, and P. A. Boncz. Sciborq: Scientific data management with bounds on runtime and quality. In *CIDR*, 2011.

[30] M. Stonebraker, J. Becla, D. J. DeWitt, K. Lim, D. Maier, O. Ratzesberger, and S. B. Zdonik. Requirements for Science Data Bases and SciDB. In *CIDR*, 2009.

[31] A. S. Szalay, G. Bell, J. Vandenberg, A. Wonders, R. Burns, D. Fay, J. Heasley, T. Hey, M. Nieto-Santisteban, A. Thakar, C. van Ingen, and R. Wilton. GrayWulf: Scalable Clustered Architecture for Data Intensive Computing. In *HICSS*, 2009.

[32] A. Vlachou, C. Doulkeridis, K. Nørvåg, and M. Vazirgiannis. On Efficient Top-k Query Processing in Highly Distributed Environments. In *SIGMOD*, 2008.

[33] D. Xin, J. Han, and K. C. Chang. Progressive and Selective Merge: Computing Top-k with Ad-hoc Ranking Functions. In *SIGMOD*, 2007.

[34] Y. Zhang, M. Kersten, M. Ivanova, and N. Nes. SciQL: Bridging the Gap Between Science and Relational DBMS. In *IDEAS*, 2011.

[35] K. Zhao, Y. Tao, and S. Zhou. Efficient Top-k Processing in Large-scaled Distributed Environments. *Data Knowl. Eng.*, 63(2):315–335, Nov. 2007.

Identifying Converging Pairs of Nodes on a Budget

Konstantina Lazaridou
Department of Informatics
Aristotle University, Thessaloniki, Greece
konlaznik@csd.auth.gr

Konstantinos Semertzidis
Computer Science and Engineering Department
University of Ioannina, Greece
ksemer@cs.uoi.gr

Evaggelia Pitoura
Computer Science and Engineering Department
University of Ioannina, Greece
pitoura@cs.uoi.gr

Panayiotis Tsaparas
Computer Science and Engineering Department
University of Ioannina, Greece
tsap@cs.uoi.gr

ABSTRACT

In this paper, we consider large graphs that evolve over time, such as graphs that model social networks. Given two instances of the graph at two points in time, we ask to identify the top pairs of nodes whose shortest path distance has decreased the most. We call these pairs *converging*. The straightforward way to address this problem is by computing the shortest path distances of all pairs at both instances and keeping the ones with the largest differences. Since for large networks this is computationally infeasible, we consider a budgeted version of the problem, where given a fixed budget of single-source shortest path computations, we seek to identify nodes that participate in as many converging pairs as possible. We evaluate a number of different approaches for our problem, that employ centrality-based, dispersion-based, and landmark-based distance estimation metrics. We also consider a classification-based approach that builds a classifier that combines the above features for predicting whether a node participates in one of the top converging pairs. We present experimental results using real-world datasets that show that we are able to identify the large majority of the top converging pairs on a very small budget.

Categories and Subject Descriptors

H.2 [**Database Management**]: Applications; E.1 [**Data Structures**]: Graphs and networks

General Terms

Algorithms, Experimentation, Performance

Keywords

graphs, top-k, shortest paths

1. INTRODUCTION

A variety of natural or man-made complex systems can be modeled as networks. Prominent examples include the Internet, the Web, transportation networks, online social networks such as Facebook, Twitter and LinkedIn, biological systems such as protein interactions or metabolic pathways, the global economy, and many more. All these systems consist of individual entities that are interconnected to form a complex system. Understanding them is not possible without understanding the underlying network. The network carries a significant amount of information about the functionality of the system as a whole as well as for its individual entities [9].

A central piece of information revealed through the network is that of *proximity*. Using the network structure, we can determine how close two individuals are in a social network, or how easy it is to navigate between different web pages. There are several ways of measuring proximity, or similarity, within the network (see, for example [6] for a recent survey). A commonly used proximity measure is the length of the shortest path distance between two nodes in the graph. Although simple, the shortest path distance is still the first notion of proximity we want to compute when examining the relationship of two individual entities. In many cases, it also provides an actionable piece of information: It is the path by which we want to route information between two individuals; the path we want to follow when moving from one place to another in a traffic network; the quickest way to create a connection within a professional network.

A succinct characteristic of real-life networks is their continuous evolution. New nodes and edges are added, often at an exponential pace. Even when observing a fixed set of nodes, their relationship can change dramatically by the addition of new edges among nodes in the set, or to other nodes outside the set. As a result, new shortest paths are created and the relationships between individuals are constantly updated.

Identifying the pairs of nodes that came the closest to each other is important in our understanding of the network, and it may be crucial for some applications. For example, in social networking sites such as *Facebook* or *LinkedIn*, if two distant users come closer over time, this could imply the appearance of similar interests or activities between them.

10.5441/002/edbt.2015.28

Hence, this further knowledge can help in making more suitable friendship recommendations. In economic networks, the decrease of the shortest path between two major players could signal a change of strategy, or have future implications for their growth. In a criminal or terrorist network, it is critical to know which suspects have come closer to each other; such moves may be indications of future actions or coalitions.

There could be also an application of this problem in protein-protein interaction networks, where the nodes are proteins within a cell and they are connected by edges if there is a possible interaction between them [3]. As new experiments reveal new connections, for two given proteins, the knowledge that they came closer together in the graph makes them candidates for an upcoming interaction. Furthermore, if a certain protein comes closer to multiple others, they may be part of the same community [12], with all underlying proteins having the same specific function within the cell.

In this paper, we address the following problem. Given two snapshots of an evolving network, we ask for the k pairs of nodes whose shortest path distance has decreased the most between the two snapshots. We call such pairs of nodes *converging*. There is a simple polynomial-time solution to our problem: Compute the all-pair shortest path distances in the two graph instances and find the pairs with the largest distance change. However, even if we used fast algorithms that approximate shortest path computation, e.g., [20], it would still require time quadratic in the size of the graph for just producing the pairs. Given that real graphs have size in the order of thousands or millions, we need solutions that scale linearly with the size of the graph.

Despite its significance, the problem of identifying the top-k converging pairs has received limited attention. The only related work we are aware of is the work in [14] which provides an approach based on the endpoints of the new edges in the second snapshot. In this paper, we address the problem from a different perspective, by predicting the nodes of the converging pairs. We formally define good candidate nodes as those nodes that belong to a *cover* of the set of the top-k converging pairs. We also introduce a novel budget formulation of the problem, where to achieve the required result, we are given a fixed budget of $m \ll n$ single-source shortest-path computations. We propose a suite of algorithms for identifying good candidate nodes based purely on the structural properties of the two graph instances. These algorithms are based on node centrality, node dispersion and landmark-based distance estimations. Then, we build a classifier that combines the proposed algorithms to effectively identify the approach that is the most appropriate for each setting. By further extending the classifier to include features of the graph, such as the graph density, we were able to build a "global" classifier that works on any graph. The classifier takes advantage of our novel method of characterizing good candidate nodes.

Our experiments with four real datasets show that we can identify most of the converging pairs with a budget equal to a very small percentage of the graph nodes. For example, for the *Internet links* dataset, with a budget of just 0.5% of the nodes, we are able to locate over 90% of the top-k converging pairs for various values of k. Furthermore, our classifiers are successful in identifying converging pairs and match the performance of the best algorithm for each dataset.

In summary, in this paper we make the following contributions:

- We formalize the problem of finding the top-k converging pairs in a graph under budget constraints as a problem of finding a vertex cover of an appropriately defined graph over the set of these pairs.
- We propose a suite of methods for identifying candidate nodes that best cover the set given a fixed budget of shortest path computations.
- We show how to combine our techniques into a single algorithm that makes use of all the proposed features.
- We perform extensive experiments that show that our techniques work well in practice, being able to find the top-k converging pairs with only a small number of shortest path computations. Compared to the approach in [14] our algorithms are more effective in identifying converging pairs, while having strong budget guarantees.

The rest of this paper is structured as follows. In Section 2, we present related work. In Section 3, we introduce the problem of identifying the top-k converging pairs and formulate it as a vertex cover problem. In Section 4, we present a suite of algorithms for identifying candidate nodes for the converging pairs. Experimental results are presented in Section 5. Finally, in Section 6, we provide conclusions and directions for future work.

2. RELATED WORK

Dynamic social network analysis has received a lot of attention, including research on community evolution, e.g., [2], and on graph generation models, e.g., [15]. In this paper, we focus on a different aspect. Given two graph instances, we ask what are the pairs of nodes that came closer to each other. This problem is different from the problem of incrementally maintaining shortest path distances in dynamic graphs, e.g., [7, 23]. Here we just want to identify the k pairs of nodes whose shortest path distance has changed the most, without re-estimating the distances for all pairs. Recent work also addresses the problem of monitoring the proximity of nodes in bipartite time-evolving social graphs [21]. The authors propose pre-computing and storing all pair distances for a small number of nodes so as to incrementally update distances and maintain the top-k most closely connected pairs, or the most central nodes. In contrast here, we consider general graphs (non bipartite) and search for the top-k pairs with the largest decrease in their shortest path distance.

Another line of research addresses graph augmentation problems that ask to find a set of edges to add to a graph so as the graph satisfies specific properties including ones involving shortest path distances. In the context of social networks, recent work considers augmentation problems within the context of improving content propagation. The authors of [22] ask which edges to add or delete in a graph so as to improve the information dissemination process and in particular to increase the leading eigenvalue of the adjacency matrix of the graph. Other recent work addresses the problem of recommending edges to add so as to maximize content spread but in addition ensure that the recommendations are also relevant [4]. The authors of [18] study the problem of selecting a k-size subset of the non-existing or ghost edges of a graph such that if they are added to the graph will minimize the average all pairs shortest path distances. The authors of [19] study the same problem as in [18] but the added edges are selected from a given set of candidate edges (e.g., the edges added between two snapshots). Different to this work, we do not assume that we can select edges to decrease the shortest paths, but rather that edges have already been added as part of the evolution of the network, and we want to find the pairs that were most affected.

The work most closely related to ours is that of [14] that studies essentially the same problem. However, their work does not impose a budget constraint on the shortest path computations. Algorithmically, their work focuses on the endpoints of new edges and relies on identifying critical edges that lie in the shortest paths of many pairs. Such notions necessitate the use of betweenness measures [9], which in general are expensive to compute. In comparison, our work introduces the idea of allocating a fixed budget of m shortest path computations, as well as the formalization of good candidate endpoints as the ones belonging to the maximum cover of the top-k converging pairs. The latter can also form the basis for building effective classifiers for the problem. We compare with the approach in [14] in our experiments.

There is a rich literature in using landmarks for shortest path estimation in large graphs, e.g., [20, 23, 25]. Various approaches for selecting landmarks have been proposed, including the ones used here. However, the problem addressed in our work is different from previous work in that we want to estimate shortest path changes rather than actual shortest paths. It is interesting to consider additional methods for selecting landmarks, such as the approach proposed in Orion [25] that maps graphs into a multi-dimensional Euclidean coordinate space, but it is beyond the scope of this work.

A different point of view in the point-to-point distance estimation problem, which bears some similarity with our approach, is considered in [5]. They propose the use of machine learning techniques for predicting the distance between two nodes of a graph. Aiming to answer real-time point-to-point distance queries, they propose to use linear functions that combine vertex-based attributes, such as the closeness centrality with landmark-based attributes, while incorporating different approaches on selecting lower bound and upper bound landmarks.

Finally, there is a large body of research on link prediction (e.g., [16]) that asks to predict which pairs of nodes will be connected in future snapshots. In this work, we are given the future snapshot, and we are interested in the efficient computation of converging pairs of nodes, not predicting direct links.

3. PROBLEM DEFINITION

We consider undirected (weighted) graphs that change over time. Let $G_t = (V_t, E_t)$ denote the graph at time instant t. As is the most common case with social networks, we consider only node and edge insertions. Thus, a dynamic graph can be seen as a sequence of slices $S_1, S_2, \dots S_t, \dots,$ of node and edge insertions. $G_t = (V_t, E_t)$ is the graph that results by aggregating all slices up until t.

For two nodes u, v, and time instant t, we use $d_t(u, v)$ to denote their shortest path distance in G_t. We call a pair of nodes $u, v \in V_t$, connected if u, v belong to the same connected component of G_t.

Now assume two graph instances, $G_{t_1} = (V_{t_1}, E_{t_1})$ and $G_{t_2} = (V_{t_2}, E_{t_2})$, with $t_2 > t_1$. Take two connected nodes u, v in G_{t_1} that are at distance $d_{t_1}(u, v)$. When considering the graph G_{t_2} the addition of new nodes and edges can only decrease the distance between u, and v. Let $\Delta_{t_1,t_2}(u, v) = d_{t_1}(u, v) - d_{t_2}(u, v)$ denote the decrease in distance between u and v. We are interested in identifying the pairs of connected nodes for which we have a sharp decrease in distance, that is $\Delta(u, v)$ is large. We focus on connected nodes since the distance between non-connected nodes is infinite, and thus, in this case, the problem comes down to finding the disconnected components that got connected.

More formally, we define our problem as follows.

Problem 1 (top-k converging pairs). *Given two graph instances G_{t_1} and G_{t_2}, at time instants t_1 and t_2, respectively, $t_2 > t_1$, and a value k, find the k connected pairs u, v of nodes in G_{t_1} with the largest $\Delta_{t_1,t_2}(u, v)$ value among all pairs of connected nodes in G_{t_1}. We call these pairs of nodes the top-k converging pairs.*

There is a simple polynomial-time solution for our problem. We compute the shortest paths of all (connected) pairs of nodes in G_{t_1} and G_{t_2} and we find the ones that have the largest decrease. Regardless of how fast we compute the shortest paths – there are fast algorithms for approximate shortest path computation, e.g., [20] – just outputting the paths for all pairs requires time $O(n^2)$, where n is the number of nodes in G_{t_1}. For networks with millions of nodes this is impractical both in terms of storage and time. We need solutions that scale linearly with the number of nodes in the graph.

To address this issue, we want to reduce the number of nodes for which we need to compute the shortest paths. We view a shortest path computation (SP computation) as a unit of computational cost, and we assume that we have a fixed computational budget that we can use for our task. This is the number m of SP computations we can perform, which is dictated by our resources and our application. We want to retrieve as many of the top-k converging pairs as possible, under the budget constraints.

We will first establish what is the minimum possible number of shortest path computations. Let P denote the set of the top-k converging pairs that we want to compute. Assume that we are given a set C such that for every pair $(u, v) \in P$, either $u \in C$, or $v \in C$. If we compute all shortest paths for the nodes in C in G_{t_1} and G_{t_2}, and we keep the top-k ones with the highest Δ_{t_1,t_2} value, then we obtain again the set P. The space and time requirements for this algorithm is $O(n|C|)$. Note that the size of the set C is at most k, so the complexity of the problem would be linear in the size of the graph. The set C is a *cover* for the set of pairs in P.

To formalize the definition of the set C, given G_{t_1}, G_{t_2} and k, we define a new graph $G_k^p = (V_1, P)$, defined over the nodes V_1 of graph G_1, $V_1 \subseteq V_{t_1}$, such that there is an edge between two nodes u and v in G_k^p, if and only if, (u, v) is a top-k converging pair. The set C described above is a *vertex cover* of the graph G_k^p, since for every edge $(u, v) \in P$ of G_k^p at least one of the endpoints of the edge belongs to C. That is, every edge is *covered* by at least one vertex. Given a vertex cover of the graph G_k^p, we can obtain the set P of the top-k converging pairs efficiently.

Obviously, when tackling Problem 1, we do not have access to graph G_k^p, or its cover, i.e., the set C. Actually, even if we knew graph G_k^p, obtaining a vertex cover C of minimum size is an NP-hard problem. However, using G_k^p, we can now reformulate our problem as follows.

Problem 2 (budgeted path cover). *Given two graph instances G_{t_1} and G_{t_2}, at time instants t_1 and t_2 respectively, with $t_2 > t_1$, a value k, and a budget m, find a set of nodes $M \subseteq V_{t_1}$ of size m such that the number of edges of G_k^p covered by M is maximized.*

We note again that we do not have access to the graph G_k^p. Problem 2 is a harder version of Problem 1, where our computational budget is limited. However, the definition of Problem 2 dictates our approach for solving Problem 1. Our goal is to identify a set of candidate nodes M that are likely to be in the cover of G_k^p. Towards this end, we have a fixed computational budget which is defined by the number m of nodes for which we can compute the single-source shortest paths distances in G_{t_1} and G_{t_2}.

Note that even if we had access to the graph G_k^p finding the minimum vertex cover, or the set of m nodes that maximize coverage is an NP-hard problem. However, it is known [24] that the greedy algorithm that each time selects the node that covers the largest number of the uncovered edges provides a solution with a logarithmic approximation ratio, that works well in practice. The greedy algorithm also has an approximation ratio for the *max-coverage* problem, where given a budget of m vertices we want to find the ones that maximize the coverage of edges. In our experiments, when we want to compare against a "good" solution, we use the vertex cover produced by the greedy algorithm. We will often refer to the greedy solution as the cover of the G_k^p graph.

4. ALGORITHMIC TECHNIQUES

We now outline a generic algorithm for finding the top-k converging pairs. As an intermediate step, our algorithm addresses Problem 2, finding a set M of nodes that cover the largest number of the top-k converging pairs. We propose a suite of algorithms for solving Problem 2. In the following, we shall use the term candidate nodes and candidate endpoints interchangeably to denote the nodes in M.

4.1 Finding the Top-k Converging Pairs

In Algorithm 1, we describe a generic algorithm for finding the top-k converging pairs. The algorithm relies on the function ComputeCandidateEndpoints that returns a set M of candidate endpoints of size m. We discuss the candidate generation below. The number of candidate endpoints m is small and thus it is feasible to compute all shortest path distances between M and the nodes in V_{t_1} in graphs G_{t_1} and G_{t_2}. Given these distances, we can compute the $\Delta_{t_1,t_2}(u, v)$ values for the pairs in $M \times V_{t_1}$ and select the top-k pairs with the highest values.

4.2 Candidate Endpoint Generation

We now describe the algorithms for generating candidate endpoints, that is, identifying nodes that best cover the top-k converging pairs, i.e., the edges of G_k^p. Our algorithms take as input the two graph instances, G_{t_1} and G_{t_2}, and the value m, and produce as output a set $M \subset V_{t_1}$ of m nodes. We want to select M such that M covers the largest number of the top-k converging pairs. The size m of M is determined by our resource budget: it is the number of nodes for which we can afford to compute the single-source shortest paths in G_{t_1} and G_{t_2}.

Algorithm 1 Generic top-k Algorithm.

```
Input: Graph snapshots $G_{t_1} = (V_{t_1}, E_{t_1})$, $G_{t_2} = (V_{t_2}, E_{t_2})$,
    $k$, $m$
Output: The set of the top-$k$ converging pairs
1: $M \leftarrow$ ComputeCandidateEndpoints($G_{t_1}$,$G_{t_2}$, $m$)
2: $D_1 \leftarrow m \times n$-dimensional array with shortest path distances in $G_{t_1}$ between the nodes in $M$ and $V_{t_1}$.
3: $D_2 \leftarrow m \times n$-dimensional array with shortest path distances in $G_{t_2}$ between the nodes in $M$ and $V_{t_2}$.
4: $\Delta_{t_1,t_2} \leftarrow D_1 - D_2$.
5: return the top-$k$ pairs $(i, j)$ with the highest values in $\Delta_{t_1,t_2}$.
```

We consider the following approaches in selecting the nodes for the set M:

- **Centrality-based:** In this case, we select nodes based on their degree or the change in their degree.
- **Dispersion-based:** In this case, we select nodes that are highly dispersed in the graph G_{t_1}, that is, they are far apart from each other. These nodes are likely to participate in a large number of top-k converging pairs, since they were far apart in the first place.
- **Landmark-based:** In this case, we select a small sample $L \subset V_{t_1}$ of the nodes in G_{t_1} to act as *landmarks*. We select the candidate endpoints based on the changes of their shortest path distances from these landmarks.
- **Hybrid:** In this case, we use again a landmark-based approach, but instead of sampling randomly the set of landmarks L, we use a dispersion-based approach to guide our choice.
- **Classification-based:** In this case, we use features provided from the previous algorithms to build a classifier that predicts whether a node is a good candidate endpoint.
- **The Incidence family of algorithms:** In this case, we consider some of the algorithms in [14] for selecting the candidate endpoints.

We now discuss each of the above approaches in detail.

4.2.1 Centrality-based selection

With this approach, we use the centrality of nodes in the graph G_{t_1} to guide us in the selection of candidate nodes. The motivation is that nodes central in graph G_{t_1} are likely to be part of the shortest paths for many nodes. Thus they seem a reasonable choice as candidate endpoints. Furthermore, we consider also nodes that had a large increase in their centrality, either in absolute terms or relative to the original value.

A simple and easy to compute centrality measure is the node degree. Formally, let $\deg_t(u)$ denote the degree of node u in graph G_t. The algorithm DEGREE ranks nodes in descending order of $\deg_{t_1}(u)$ and selects the top-m nodes. The algorithm DEGDIFF ranks nodes in descending order of $\deg_{t_2}(u) - \deg_{t_1}(u)$ and selects the top-m nodes. Finally, the DEGREL ranks nodes in descending order of $(\deg_{t_2}(u) - \deg_{t_1}(u))/\deg_{t_1}(u)$ and selects the top-m nodes.

4.2.2 *Dispersion-based selection*

With this approach, we want to select as candidates the m nodes in G_{t_1} that are the furthest apart from each other. That is, we want M to contain the most dispersed set of nodes. We define this in two different ways.

The first approach is to select nodes such that the average distance between all pair of selected nodes is maximized:

$$M = \arg \max_{\substack{S \subseteq V_{t_1} \\ |S|=m}} \frac{1}{\binom{m}{2}} \sum_{v_i, v_j \in S} d_{t_1}(v_i, v_j) \qquad (1)$$

This approach tends to select nodes in the perimeter of the graph.

Alternatively, we can select nodes such that the minimum distance between any two pairs of the selected nodes is maximized:

$$M = \arg \max_{\substack{S \subseteq V_{t_1} \\ |S|=m}} \min_{v_i, v_j \in S} d_{t_1}(v_i, v_j) \qquad (2)$$

This approach tends to select nodes that "cover" the graph.

Even if we had the pairwise distances between all nodes, finding the optimal set of nodes for both cases is an NP-hard problem (for example, see [13] for a recent discussion on this topic). We thus use a greedy algorithm that at each step selects the node that maximizes the dispersion with respect to the nodes selected so far. We refer to the algorithm that maximizes the average distance as MAXAVG and to the algorithm that maximizes the minimum distance as MAXMIN.

4.2.3 *Landmark-based selection*

With this approach, we make use of a set $L \subset V_{t_1}$ of ℓ nodes that we call *landmarks* and compute the distances of all nodes in G_{t_1} and G_{t_2} from the nodes in L. We select our candidates based on how close they came to the landmarks in graph G_{t_2}.

Specifically, let $L = (w_1, \ldots, w_\ell)$ be an ordered set of landmark nodes of graph G_{t_1}, with $\ell \ll |V_{t_1}|$. We associate with each node $u \in V_{t_1}$, two ℓ-dimensional vectors, $DL_1(u)$ and $DL_2(u)$, where $DL_1(u)[i] = d_{t_1}(u, w_i)$ and $DL_2(u)[i] = d_{t_2}(u, w_i)$. The vector $\Delta L_{t_1,t_2}(u) = DL_1(u) - DL_2(u)$ captures the change in the shortest path distance from u to the landmarks in L.

We now select as candidate endpoints the nodes that came "closest" to the landmarks L in the graph G_{t_2}. To measure the degree of distance change, we use the L_1 and L_∞ norms of the vector $\Delta L_{t_1,t_2}(u)$.

$$\|\Delta L_{t_1,t_2}(u)\|_1 = \sum_{i=1}^{\ell} \Delta L_{t_1,t_2}(u)[i] \qquad (3)$$

$$\|\Delta L_{t_1,t_2}(u)\|_\infty = \max_{i=1}^{\ell} \Delta L_{t_1,t_2}(u)[i] \qquad (4)$$

Table 1: Shortest-path computations for different approaches

Approach	Candidate Generation	top-k Pairs
Degree-based	0	$2m$
Dispersion-based	m	m
Landmark-based	2ℓ	$2m - 2\ell$
Hybrid	2ℓ	$2m - 2\ell$
Classification-based	$3 \cdot 2\ell$	$2m - 3 \cdot 2\ell$

In the SUMDIFF algorithm, we select the m nodes with the largest L_1-norm, while in the MAXDIFF algorithm, we select the m nodes with the largest L_∞-norm. The intuition is that these are nodes that became more central in the graph G_{t_2} and thus are likely to participate in many changed shortest paths. The SUMDIFF algorithm is somewhat related in motivation to the greedy algorithm for finding the minimum vertex cover. The node with the largest L_1-norm value is the node that covers the most of the distance changes in the $(\ell \times n)$-matrix $\Delta L_{t_1,t_2}$.

4.2.4 *Hybrid selection*

With the hybrid approach, we attempt to get the best of both worlds by combining the dispersion-based approach with the landmark based approach. Specifically, we use the dispersion-based techniques to select the ℓ landmarks, and then apply the landmark based approach.

This hybrid approach is motivated by two factors. First, in order to obtain the landmark distances in the graphs G_{t_1} and G_{t_2}, we need to pay a cost of ℓ shortest path computations in each graph. When selecting a set of random landmarks this cost comes with no payoff since it is unlikely that the randomly selected nodes will cover any of the converging pairs. Using the nodes selected from the dispersion based algorithm guarantees that we will obtain some benefit from the landmark selection. Second, intuitively, it seems that dispersed nodes should work better as landmarks since they cover different parts of the graph, and thus we can better capture the fact that a node came closer to some part of the graph.

Depending on the algorithm that we use for the landmark selection policy, and the norm we use for measuring the change in the distance to the landmarks, we get four different algorithms, namely the MAXAVG-SUMDIFF (MASD), MAXAVG-MAXDIFF (MAMD), MAXMIN-SUMDIFF (MMSD) and MAXMIN-MAXDIFF (MMMD) algorithms.

4.2.5 *Classification-based selection*

Our goal in the candidate endpoint generation is to identify nodes that are likely to cover the largest number of converging pairs. We can think of most of the previous algorithms as methods for identifying features that characterize good candidates. A natural extension of the above approach is to combine all these features into a single algorithm using a classifier that will try to predict whether a node is a "good" endpoint or not. The benefit of such an approach is that it combines multiple features instead of one, and it automatically finds the appropriate features to use for each dataset without manual inspection. However, we need to allocate resources for training the classifier.

An important question in this setting is how to determine the positive class for the classifier. What do we mean by

Table 2: Dataset Characteristics

Dataset	No of nodes		No of edges		diameter		max Δ_{t_1,t_2}	not-connected
	G_{t_1}	G_{t_2}	G_{t_1}	G_{t_2}	G_{t_1}	G_{t_2}		G_{t_1}
Actors	1,851	1,886	45,584	56,981	5	5	3	0
Internet links	21,835	25,526	83,857	104,824	12	11	6	80
Facebook	4,436	4,734	25,197	31,498	12	11	7	27
DBLP	15,391	17,992	38,866	48,618	17	15	9	3,864

"good" endpoint? For this task we make use of the vertex cover of the graph G_k^p, that we obtain with the greedy algorithm. This is a collection of nodes that concisely cover the changed paths, and thus it is reasonable to use them as the positive class for our classification task. As features, we use features employed by our algorithms such as the degree of the nodes in G_{t_1}, and G_{t_2}, the degree difference, and the relative degree difference, the L_1 and L_∞ norm of the distances to random landmarks and landmarks selected accordingly to the MaxMin and MaxAvg algorithms. We train one such classifier for each dataset as described in the experiments. We refer to this classifier, as the *local* classifier of the dataset.

We also consider a classifier that can operate on *any* dataset. For this classifier, we extend the feature set with features characteristic of the dataset. In particular, we consider the density, and the maximum degree of the two graph snapshots. The resulting classifier, termed *global* classifier, once trained, can be used to generate candidate endpoints for the top-k converging pairs for any two snapshots of any graph.

For the classification, we use logistic regression. The benefit of the logistic regression algorithm is that it outputs a probability for a node to belong to a given class, in our case to the vertex cover. We sort the nodes in decreasing order of this probability and we output the top-m nodes.

4.2.6 *Incidence Algorithm*

To the best of our knowledge, the first paper that addresses the problem of identifying the top converging pairs in evolving social networks, is [14]. In this paper, the nodes that receive new edges in the second timestamp are called *active* and are considered as the most probable nodes to participate in the top converging pairs. The *Incidence Algorithm* is introduced, where after constructing the set of the *active* nodes A, a number of $|A|$ shortest path computations is performed on both instances of the graph, in order to obtain the pairs with the maximum distance difference.

There are two variations of the *Incidence Algorithm*. In the first one, called *Selective Expansion*, the neighbors of the endpoints in A are also considered as candidates. Every neighbor is evaluated according to the number of its *important* edges. For this purpose, the notion of edge *importance* in a social network is introduced, which is an estimate of the edge betweenness centrality, computed using a randomly selected set of shortest path trees. In our experiments, we used the actual edge betweenness centrality, giving an advantage to the *Incidence* algorithm. In this variation, the algorithm proceeds iteratively, inserting to A new neighbors, and executing *Incidence*, until there are no more new pairs discovered.

The second variation of the *Incidence Algorithm* proposes a number of rank strategies for the *active* nodes, in order to choose the top few of them that are likely to participate in the converging pairs. The rank policies are divided to degree-based and betweeness-based. Among other strategies concerning weighted social graphs (which we do not consider in our work), the authors of [14] rank the *active* nodes using four different policies: their degree in G_{t_2}, their degree difference between the two graph instances, the sum of the *importance* of the edges that a node received in G_{t_2} and finally the difference of the last measure among G_{t_1} and G_{t_2}.

4.3 Complexity

Recall that the value m is not only the number of candidate endpoints, but also the computational budget we have in terms of time and space, that determines how many single-source shortest-path computations we can afford to perform on graphs G_{t_1} and G_{t_2}. Therefore, all algorithms perform exactly $2m$ shortest path computations. Table 1 demonstrates how these computations are allocated in the different phases of the algorithm, for different selection policies. The first phase concerns the shortest path computations required for selecting the candidate endpoints. The second phase involves the computation of the single-source shortest-paths for the candidate endpoints, in both snapshots. For the generated pairs, we compute the decrease in their shortest paths, and select the k ones whose distance decreased the most.

Note that the dispersion based methods need to compute shortest paths only in the graph G_{t_1}, in order to select the candidate endpoints. We still need to compute the shortest paths on G_{t_2} though, in order to output the top-k pairs. Also the classifier requires $3 \cdot 2\ell$ shortest path computations for computing the landmarks in three different ways, in order to produce the features.

5. EXPERIMENTAL EVALUATION

In this section, we compare the performance of the various algorithms and classifiers in terms of identifying the actual top-k converging pairs for various values of the available budget m.

5.1 Datasets and Setting

To evaluate the performance of our algorithms, we need to be able to compute the true top-k converging pairs. Therefore, we use datasets of manageable size, for which it is feasible to compute all-pairs shortest paths. We consider the following four real datasets:

- The *Actors* dataset, where the nodes are actors and there is a connection between two film actors if they both appeared in the same movie. The dataset was obtained from the IMDB web site[1], and it spans the years from 1998 to 2010.

[1]http://www.imdb.com

Table 3: Characteristics of the G_k^p graphs (number of nodes and edges) and their maximum vertex cover (number of nodes).

Dataset	$i = 0$			$i = 1$			$i = 2$		
Actors	$\delta = 3$			$\delta = 2$			$\delta = 1$		
	endpoints	pairs	maxcover	endpoints	pairs	maxcover	endpoints	pairs	maxcover
	35	27	10	1,350	4,081	446	1,851	202,899	9
Internet links	$\delta = 6$			$\delta = 5$			$\delta = 4$		
	endpoints	pairs	maxcover	endpoints	pairs	maxcover	endpoints	pairs	maxcover
	28	46	8	382	734	41	9,196	17,896	194
Facebook	$\delta = 7$			$\delta = 6$			$\delta = 5$		
	endpoints	pairs	maxcover	endpoints	pairs	maxcover	endpoints	pairs	maxcover
	4	2	2	44	37	16	409	591	60
DBLP	$\delta = 9$			$\delta = 8$			$\delta = 7$		
	endpoints	pairs	maxcover	endpoints	pairs	maxcover	endpoints	pairs	maxcover
	6	4	2	68	68	12	289	462	46

- The *Internet links* dataset is an undirected graph representing the AS-level connectivity of the Internet [17, 10] (an Autonomous System (AS) represents a single administrative domain on the Internet). Each node in the graph is an AS and an edge between two nodes represents a message that was exchanged between them, using the inter-domain routing protocol.

- The *Facebook* dataset, where nodes are users of Facebook, and edges between two nodes denote friendship. The dataset contains 31,498 connections, which were created sequentially in 31,498 different time points.

- The *DBLP* dataset, where nodes are authors and there is an edge between two authors if they wrote a paper together. The dataset was obtained from the *DBLP* site[2] and includes articles of 14 top conferences in data mining, databases, theory and the WWW from 1983 to 2013.

Each dataset was divided into two snapshots, so that the initial snapshot, G_{t_1}, contains 80 percent of the edges, and the second snapshot, G_{t_2}, contains the entire graph. Table 2 provides a summary of the characteristics of each dataset.

In selecting the values of k on which to evaluate our algorithms, we note that for any given k, there are many ties, that is, there are many pairs with the same shortest path change. This means that there are many different sets of top-k converging pairs and G_k^p graphs. We set k to a value that guarantees a single optimal solution. Specifically, for two graph instances G_{t_1} and G_{t_2}, let Δ be the maximum distance decrease over all connected pairs of nodes in G_{t_1}. We test our algorithms by assigning values to k that correspond to the number of pairs whose distance change is at least equal to δ, where δ takes values Δ, $\Delta - 1$, and $\Delta - 2$. Setting k as above makes the problem harder, since there is a single optimal solution that we must retrieve that includes *all* converging pairs with shortest path distance change at least δ. For smaller values of the given k, our algorithms work even better, since in this case, retrieving any of (many) tying pairs suffices.

In Table 3, we report for each dataset the number of pairs whose path distance change is at least $\delta = \Delta - i$, for $i = 0, 1, 2$, the number of distinct endpoints involved in these pairs, and the size of the maximum cover as computed by the greedy algorithm. For example, for the DBLP dataset, for $\delta = 8$ ($\Delta - 1$), there are 68 pairs whose distance was reduced by at least 8. These pairs involved 68 distinct nodes, and they can be covered with 12 nodes. When running our algorithms we set $k = 68$, that is, we look for the top-68 converging pairs.

The main parameter of our algorithms is the available budget expressed through parameter m, i.e., the number of candidate endpoints for which we can compute shortest paths. The performance of an algorithm is measured in terms of *coverage*: The percentage of top-k converging pairs that are retrieved by the algorithm. Note that these are pairs with at least one endpoint in the candidate set produced by the algorithm. The goal of the experiments is to study the cost-coverage tradeoff. We want to understand how the coverage grows as we increase the budget, and the maximum coverage we can obtain with a small budget ($m = 100$). For simplicity, we fix the number ℓ of landmarks to 10 for all algorithms. A larger number of landmarks did not improve the performance.

In Table 4 we give an overview of the algorithms we consider, and a quick index for the algorithm names.

5.2 Single Feature Algorithms

In this experiment, we evaluate the performance of the proposed single-feature algorithms (all algorithms, except for the classification-based algorithms). Our evaluation is in terms of coverage of the top-k converging pairs.

We first evaluate the coverage of our algorithms (percentage of top-k converging pairs found) for a fixed budget $m = 100$ and various values of k. Table 5 reports the coverage of converging pairs with distance change at least δ for various δ, which corresponds to a number of different k values, ranging from $k = 2$ for Facebook and $\delta = 7$ to $k = 202{,}899$ for Actors and $\delta = 1$. The bold entries correspond to the best performing algorithm for a particular input.

As shown, the *centrality-based algorithms* (based on degree) achieve the lowest coverage almost always for all datasets but for *Actors*. The algorithm that orders the candidate endpoints according to their degree in the original graph has actually almost zero coverage for all datasets, indicating that degree is negatively correlated with participation to changed shortest paths. It appears that nodes with very high degree are already central in the graph, and thus most of their shortest paths are already short. Surprisingly the degree difference is also an ineffective feature for selecting good candidates. This can be explained from the prefer-

[2]http://dblp.uni-trier.de/

Table 4: Overview of Candidate Selection Algorithms.

DEGREE	Selects the m nodes with the largest $\deg_{t_1}(u)$.
DEGDIFF	Selects the m nodes with the largest $\deg_{t_2}(u) - \deg_{t_1}(u)$.
DEGREL	Selects the m nodes with the largest $(\deg_{t_2}(u) - \deg_{t_1}(u))/\deg_{t_1}(u)$.
MAXMIN	Selects greedily m nodes in the first snapshot, such that each new node maximizes the minimum distance to the already selected nodes.
MAXAVG	Selects greedily m nodes in the first snapshot, such that each new node maximizes the average distance to the already selected nodes.
SUMDIFF	Selects the m nodes with the largest sum of distance decreases from a set of random landmarks L.
MAXDIFF	selects the m nodes with the largest maximum distance decrease from a set of random landmarks L.
MMSD	MAXMIN-SUMDIFF: Uses MAXMIN for landmark selection, and SUMDIFF for selecting the m nodes.
MMMD	MAXMIN-MAXDIFF: Uses MAXMIN for landmark selection, and MAXDIFF for selecting the m nodes.
MASD	MAXAVG-SUMDIFF: Uses MAXAVG for landmark selection, and SUMDIFF for selecting the m nodes.
MAMD	MAXAVG-MAXDIFF: Uses MAXAVG for landmark selection, and MAXDIFF for selecting the m nodes.
INCDEG	Selects the m of the *active* nodes with the largest $\deg_{t_2}(u) - \deg_{t_1}(u)$ [14].
INCBEET	Selects the m of the *active* nodes with the largest increase in the total betweenness of their incident edges [14].

ential attachment principle [1]: nodes with high degree are more likely to obtain new links. We indeed observed strong correlation between degree and degree change. Relative degree difference mitigates the effect of high degree to some extent, and this is why it performs better than degree and degree difference on all datasets. Still it underperforms compared to other algorithms. The poor performance of the centrality-based algorithms indicates that approaches based on the endpoints of the new edges as the one in [14] are less efficient than selecting candidate nodes based on other features.

The exception to the above observations is the *Actors* dataset. For this dataset, DEGREL is among the best algorithms. We note that this is a dense dataset where many of the top changed shortest paths are reduced to single edges. In this setting, the addition of new edges to a node has a stronger effect on the shortest path changes.

The *dispersion-based algorithms* are relatively successful in discovering the converging pairs. MAXAVG outperforms MAXMIN in almost all cases. MAXAVG gives preference to nodes in the outskirts of the graph while MAXMIN tends to select nodes that cover the different clusters in the graph [8]. Peripheral nodes are more likely to come significantly closer to other nodes in the graph. The satisfactory performance of the dispersion-based algorithms and the fact that they do not require knowledge of the second snapshot indicates that dispersion techniques could also be used as predictors of converging pairs.

From the two *landmark-based algorithms*, SUMDIFF works consistently better than MAXDIFF. SUMDIFF considers nodes that came closer to many landmarks and thus discovers nodes that become central in the new graph. We think of the SUMDIFF algorithm as a sampling of the distance changes of the nodes in the graph. Nodes with high SUMDIFF value are likely to have come close to many nodes in the graph. Thus, in some sense, SUMDIFF is trying to approximate the methodology of the greedy algorithm for finding a vertex cover for G_k^p.

Finally, among the *hybrid algorithms*, the best coverage is attained in most cases by MAXMIN-SUMDIFF (MMSD). The MAXMIN-SUMDIFF algorithm exploits the ability of the MAXMIN algorithm to select representative landmarks that cover the initial graph and the ability of SUMDIFF to select nodes that come closer to all these representative nodes. Note that although in general MAXAVG is a better dispersion algorithm than MAXMIN, MAXMIN-based landmark selection tends to outperform MAXAVG-based selection. This is reasonable, since, for landmark selection, it is better to select nodes that cover the graph rather than peripheral nodes.

We now study the coverage achieved for different values of the budget m. Figure 1 shows the coverage achieved by the landmark-based algorithms for various values of m for all datasets. In general, the algorithms based on SUMDIFF converge faster confirming our intuition that they cover many pairs. The plot also demonstrates that landmark-based algorithms waste part of their computational budget in computing shortest path distances for the $l = 10$ random nodes selected as landmarks that are not likely to be endpoints. Thus, these algorithms obtain no coverage for this computation. On the other hand, the hybrid algorithms benefit from the fact that they choose meaningful candidates as landmarks, and as a result the effort for the landmark shortest path computations is not wasted. Also, notice that MASD and MMSD attain 90% coverage for m smaller than 50.

We also look into the set of candidate endpoints generated by our algorithms to see to what extent this set consists of (1) nodes in G_k^p and (2) nodes in the vertex cover of G_k^p produced by the greedy algorithm. We refer to the second set as greedy-cover. For this experiment we use the Facebook dataset and $\delta = 6$ (k = 37). We study the best-performing algorithms for this dataset: the landmark-based and hybrid algorithms. Figure 2(a) reports the percentage of the candidate nodes that belong to G_k^p and Figure 2(b) the percentage that belongs to greedy-cover for various values of m. Similar behavior is noticed for the other datasets and algorithms. Not surprisingly, we observe that algorithms that cover many paths also have high intersection with both sets. It is interesting to note that the algorithms based on SUMDIFF have the largest intersection with the greedy-cover, that is, they discover high-quality candidate nodes.

5.3 Classification-based methods

In our experiments, we observed that different algorithms perform well for different datasets. A natural question is whether we can combine the individual algorithms to generate better candidate endpoints and if so, how we should weight the contribution of each individual algorithm. To this end, we view the above algorithms as features and use them

Table 5: Coverage: percentage of converging pairs found for $m = 100$.

	Actors			*Internet links*			*Facebook*			*DBLP*		
	$\delta = 3$ $k = 27$	$\delta = 2$ $k = 4{,}081$	$\delta = 1$ $k = 202{,}899$	$\delta = 6$ $k = 46$	$\delta = 5$ $k = 734$	$\delta = 4$ $k = 17{,}896$	$\delta = 7$ $k = 2$	$\delta = 6$ $k = 37$	$\delta = 5$ $k = 591$	$\delta = 9$ $k = 4$	$\delta = 8$ $k = 68$	$\delta = 7$ $k = 462$
Degree	0	3.99	5.10	0	0	0.19	0	0	0.51	0	0	0
DegDiff	37.04	37.03	22.46	0	0.27	1	0	5.41	6.77	0	7.3	5.2
DegRel	**100**	**66.16**	**34.24**	41.30	44.82	45.17	0	70.27	44.50	**100**	57.35	39.61
MaxMin	59.26	28.13	14.0	67.39	65.67	66.12	**100**	64.87	50.42	75	52.94	42.42
MaxAvg	**100**	43.23	17.03	86.96	81.88	78.65	50	86.49	**93.40**	75	58.82	47.62
SumDiff	74.07	57.59	28.34	96.96	95.05	95.07	55	93.24	93.37	75	84.12	69.83
MaxDiff	83.33	31.74	16.97	92.39	92.00	88.31	45	84.32	87.73	77.5	74,41	63.31
MMSD	96.30	56.26	27.14	**97.83**	95.1	**96.0**	50	**94.56**	90.86	75	79.41	62.34
MMMD	37.04	25.41	15.06	**97.83**	88.69	80.16	50	83.78	80.20	75	72.06	67.53
MASD	**100**	54.64	27.11	**97.83**	**95.1**	95.60	50	89.19	92.22	75	86.77	67.53
MAMD	44.44	26.32	15.06	95.65	94.41	82.09	0	89.19	81.56	75	**88.24**	**76.19**
IncDeg	37.04	37.44	22.77	0	0	0.32	0	2.7	3.9	75	45.6	32.47
IncBeet	29.63	27	15.34	0	0.41	0.73	0	16.22	8.63	0	2.94	1.73

to build a classifier to predict whether a node is a "good" candidate endpoint or not. More precisely we use the following features: the degree of the nodes in the first snapshot; the degree difference; the degree relative difference; the L_1 and L_∞ norm of the distances to random landmarks and landmarks selected accordingly to the MaxMin and the MaxAvg algorithms. All features are normalized in the interval [-1,1]. An important benefit of using a classifier is that it automatically finds the appropriate features for each dataset.

For the positive class of the classifier, that is for the class that corresponds to "good" endpoints, we use the vertex cover computed by the greedy algorithm for the graph G^p_k. As shown by our experiments, participation of a node in the vertex cover is a strong indication that a node is a good candidate. We also experimented with using all endpoints in G^p_k, and the results were very similar.

We use the *LIBLINEAR* implementation of the logistic regression classifier [11]. The logistic regression classifier outputs a probability for every node to belong to the positive class. Using this probability, we order the nodes according to their likelihood of being good endpoints. We sort the nodes in descending order of this probability and we output the top-m nodes.

For the evaluation of the classifier, we split each dataset into four snapshots. To train the classifier, we use snapshots $G^0_{t_1}$ that includes 40 percent of the edges and $G^0_{t_2}$ that includes 60 percent of the edges. To test the classifier, we use snapshots G_{t_1} that includes 80 percent of the edges and G_{t_2} that includes the full graph, that is, the same snapshots used for the evaluation of the single-feature algorithms. This allows for a direct comparison of the results. We used the same δ for both training and testing.

We consider two types of classifiers. A *local* classifier build for each dataset which we denote as L-Classifier, and a *global* classifier that can work for any dataset which we denote as G-Classifier. For the global classifier, we extract additional features from all datasets. In particular, we compute the density and the maximum node degrees of the training graph snapshots that we normalize appropriately within the interval [-1, 1]. We create the global classification model using training data from all four datasets in equal proportions.

Figure 3 compares the coverage achieved by L-Classifier and G-Classifier, with the best algorithm for each dataset. Note that the best algorithm is different for each dataset. The classification algorithm is able to automatically detect the appropriate features and produce a solution that has high coverage for each dataset. Note that the classifiers are handicapped by the set-up cost of the landmark computation (the first 30 shortest-path computations). Still, they are able to catch up with the best algorithm.

The only case where this does not happen is for the *Actors* dataset, when using G-Classifier. The poor performance can be explained by the differentiation of *Actors* dataset compared to the other datasets (Section 5.2). Since 75% of the training set for G-Classifier consists of features extracted from the *Facebook*, *Internet links*, and *DBLP* datasets, and only 25% of *Actors* features, the G-Classifier fails to produce a high coverage.

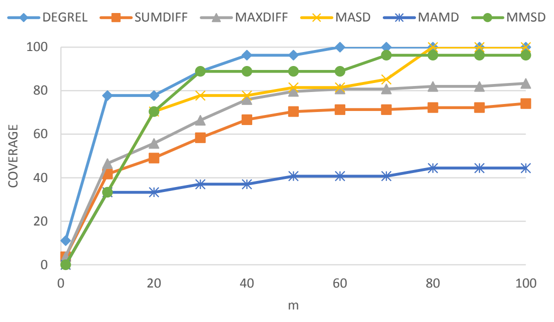
5.4 The Incidence algorithms

Finally, we compare our work to the approach in [14]. For this comparison, we implemented the original version of the Incidence algorithm, which does not use any kind of budget in the shortest path computations. This algorithm achieves very high coverage as shown in Table 6. However the set of the *active* nodes A, which is the set of nodes for which we need to compute the shortest paths, is a large fraction of the original graph. The smallest set A is computed for the *DBLP* dataset, and it is 11.66% of the G_{t_1} size ($m = 1{,}794$). In comparison our budget ($m = 100$) does not exceed 0.65% of the graph size. The largest set A is computed for the *Facebook* dataset, and it is around 66% of G_{t_1}, while our budget of $m = 100$ candidates is just 2.25% of the graph. Given the large number of candidates used by Incidence, the algorithm achieves almost complete coverage. At the same, time the time complexity is excessively high. For efficiency reasons, we did not test the effectiveness of *Selective Expansion*, as it is a recursive process that is very time consuming. It would lead us to use a very large set of candidate nodes and eventually to solve the problem by performing the baseline algorithm (computing all-pairs shortest paths), which is prohibitively expensive.

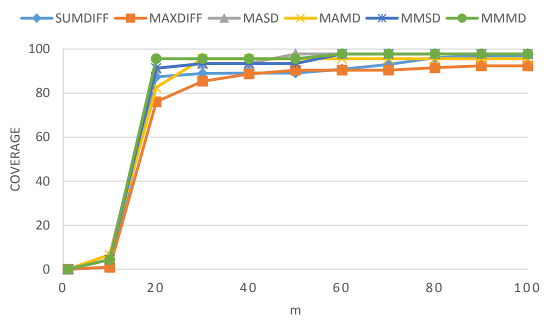
We also compare with the approach of [14] under budget constraints. Table 5 shows the best of the degree-based policies, IncDeg, where the *active* nodes are ranked by their

Table 6: The percentage of G_{t_1} that the active nodes form and the coverage of *Incidence* Algorithm.

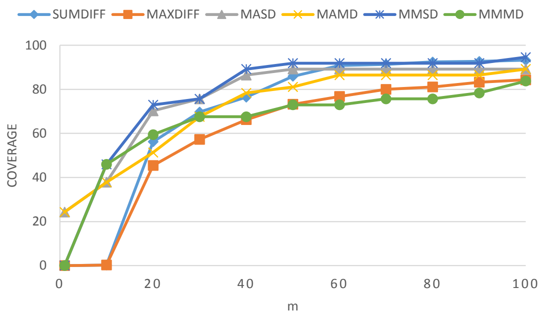
	Actors			*Internet links*			*Facebook*			*DBLP*		
	$\delta = 3$	$\delta = 2$	$\delta = 1$	$\delta = 6$	$\delta = 5$	$\delta = 4$	$\delta = 7$	$\delta = 6$	$\delta = 5$	$\delta = 9$	$\delta = 8$	$\delta = 7$
INCIDENCE active *nodes*	*m = 1,197* (64.66%)			*m = 5,071* (23.22%)			*m = 2,914* (65.7%)			*m = 1,794* (11.66%)		
INCIDENCE *coverage*	100	100	99.35	89.13	95.1	93.91	100	100	99.32	100	97.1	90.9



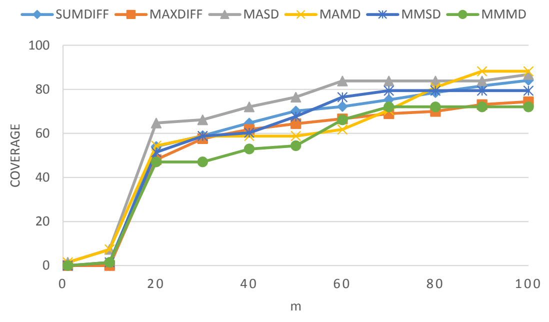
(a) *Pair coverage for Actors*



(b) *Pair coverage for Internet links*

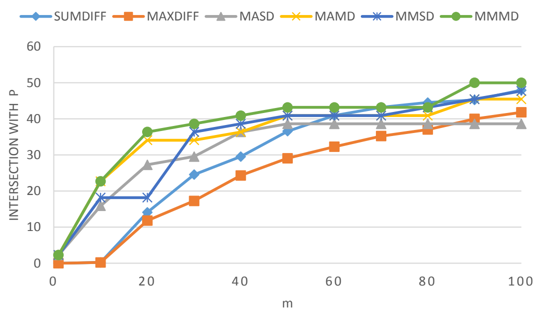


(c) *Pair coverage for Facebook*

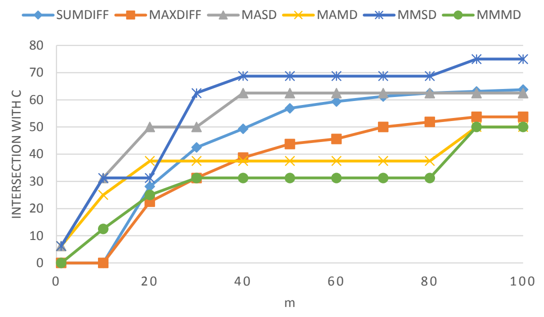


(d) *Pair coverage for DBLP*

Figure 1: Coverage of the top-k converging pairs, for the (a) *Actors* dataset and $\delta = 3$ ($k = 27$), (b) *Internet links* dataset and $\delta = 6$ ($k = 46$), (c) *Facebook* dataset and $\delta = 6$ ($k = 37$), and (d) *DBLP* dataset and $\delta = 8$ ($k = 68$).

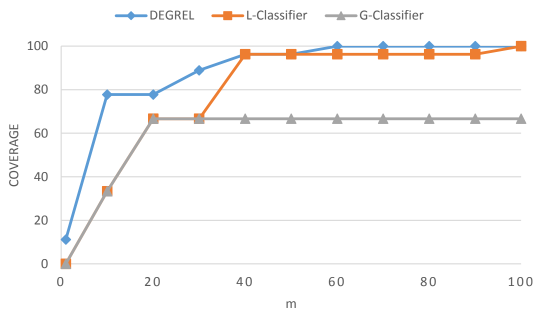


(a) *Intersection with nodes in* G_k^p

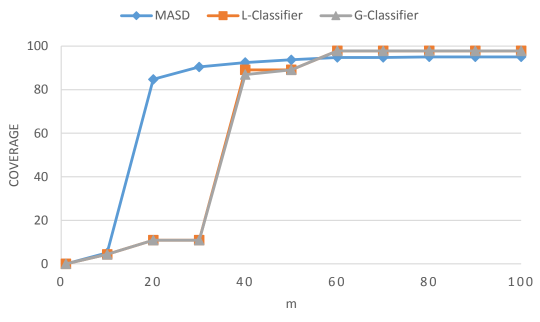


(b) *Intersection with greedy-cover*

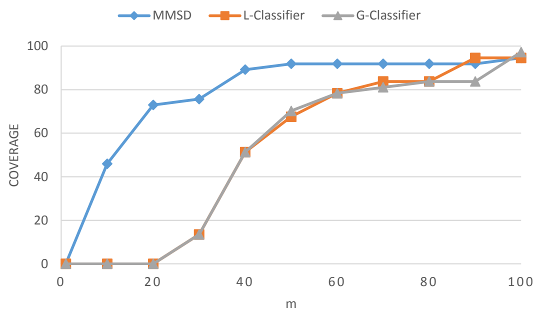
Figure 2: (a) Intersection of candidate nodes with (a) the nodes of G_k^p and (b) the greedy-cover set, for various values of m for the *Facebook* dataset and $\delta = 6$ ($k = 37$).



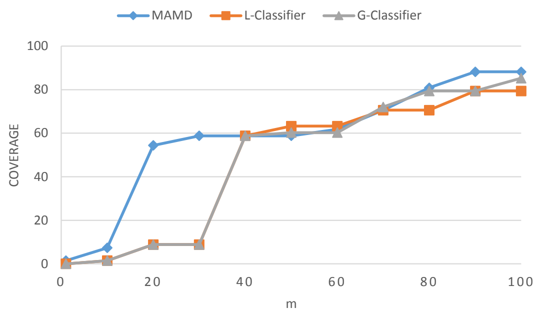
(a) *Pair coverage for Actors*



(b) *Pair coverage for Internet links*



(c) *Pair coverage for Facebook*



(d) *Pair coverage for DBLP*

Figure 3: Coverage of the top-k converging pairs, achieved by best algorithm, G-Classifier, L-Classifier trained with the vertex cover computed by the greedy algorithm, for the (a) *Actors* dataset and $\delta = 3$ ($k = 27$), (b) *Internet links* dataset and $\delta = 6$ ($k = 46$), (c) *Facebook* dataset and $\delta = 6$ ($k = 37$), and (d) *DBLP* dataset and $\delta = 8$ ($k = 68$).

degree change and IncBeet, where active nodes are ranked by the increase of the betweeness centrality of the edges incident to the nodes. We compute the actual edge betweenness centrality instead of estimating it, giving an advantage to the IncBeet algorithm. We can observe in Table 5 that IncDeg and IncBet have low performance. In particular, for the *Internet links* dataset, with $\delta = 4$ and $m = 100$ (0.5% of G_{t_1} size), MMSD achieves almost 96% coverage, while the coverage of IncBet does not exceed 1%. In almost all of our experiments, with exception the case of the *DBLP* dataset with $\delta = 9$, both of the two best variations of the Incidence algorithm underperform in term of coverage, as they can not discover more than 50% of the top converging pairs.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we focus on the problem of identifying top-k converging pairs of nodes, that is, pairs of nodes that came closer together between two snapshots of an evolving social graph. We address the problem using purely structural properties of the two graph instances. Since a brute-force method for computing all pair shortest path distances in the two instances is not cost effective, we tackle the problem from a different angle, by predicting the endpoints of such pairs. In doing so, we introduce two novel ideas: (1) allocating a fixed budget of m shortest-path computations and (2) formally defining good candidate endpoints as those belonging to the vertex cover of the top-k converging pairs. We propose a suite of algorithms for selecting candidate nodes and build a classifier that combines them to effectively identify the most appropriate algorithm for each setting. The classifier takes advantage of our novel method of characterizing good candidate endpoints.

For future work, an interesting variation of the problem to consider is the converging pair prediction task, where we are given only the initial graph snapshot and we are asked to "predict" the converging pairs. This problem can be seen as an extension of the link prediction problem which asks whether a single edge will be added between a pair of nodes. Finally, our work can be extended by considering non structural properties, such as additional attributes of the edges or nodes.

7. ACKNOWLEDGMENTS

This work has been supported by the Marie Curie Reintegration Grant project titled JMUGCS which has received research funding from the European Union, as well as the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thales which has been co-financed by the European Union (European Social Fund - ESF) and Greek national funds. Investing in knowledge society through the European Social Fund.

8. REFERENCES

[1] Réka Albert and Albert-László Barabási. Emergence of scale in random networks. *Science*, 286:509–512, 1999.

[2] Lars Backstrom, Daniel P. Huttenlocher, Jon M. Kleinberg, and Xiangyang Lan. Group formation in

large social networks: membership, growth, and evolution. In *KDD*, pages 44–54, 2006.

[3] Mario Cannataro, Pietro H. Guzzi, and Pierangelo Veltri. Protein-to-protein interactions: Technologies, databases, and algorithms. *ACM Comput. Surv.*, 43(1):1:1–1:36, December 2010.

[4] Vineet Chaoji, Sayan Ranu, Rajeev Rastogi, and Rushi Bhatt. Recommendations to boost content spread in social networks. In *WWW*, pages 529–538, 2012.

[5] M. Christoforaki and T. Suel. Estimating pairwise distances in large graphs. In *BigData*, 2014.

[6] Sara Cohen, Benny Kimelfeld, and Georgia Koutrika. A survey on proximity measures for social networks. In *SeCO Book*, pages 191–206. ACM, 2012.

[7] Camil Demetrescu and Giuseppe F. Italiano. A new approach to dynamic all pairs shortest paths. In *STOC*, pages 159–166, 2003.

[8] Marina Drosou and Evaggelia Pitoura. Disc diversity: result diversification based on dissimilarity and coverage. *PVLDB*, 6(1):13–24, 2012.

[9] David A. Easley and Jon M. Kleinberg. *Networks, Crowds, and Markets - Reasoning About a Highly Connected World*. Cambridge University Press, 2010.

[10] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. In *SIGCOMM*, pages 251–262, 1999.

[11] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874, 2008.

[12] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826, 2002.

[13] Sreenivas Gollapudi and Aneesh Sharma. An axiomatic approach for result diversification. In *WWW*, pages 381–390, 2009.

[14] Manish Gupta, Charu C. Aggarwal, and Jiawei Han. Finding top-k shortest path distance changes in an evolutionary network. In *SSTD*, pages 130–148, 2011.

[15] Jure Leskovec, Jon M. Kleinberg, and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. *TKDD*, 1(1), 2007.

[16] David Liben-Nowell and Jon M. Kleinberg. The link-prediction problem for social networks. *JASIST*, 58(7):1019–1031, 2007.

[17] Alan Mislove. *Online Social Networks: Measurement, Analysis, and Applications to Distributed Information Systems*. PhD thesis, Rice University, Department of Computer Science, May 2009.

[18] Manos Papagelis, Francesco Bonchi, and Aristides Gionis. Suggesting ghost edges for a smaller world. In *CIKM*, pages 2305–2308, 2011.

[19] N. Parotisidis, E. Pitoura, and P. Tsaparas. Selecting shortcuts for a smaller world. In *SIAM International Conference on Data Mining (SDM)*, 2015.

[20] Michalis Potamias, Francesco Bonchi, Carlos Castillo, and Aristides Gionis. Fast shortest path distance estimation in large networks. In *CIKM*, pages 867–876, 2009.

[21] Hanghang Tong, Spiros Papadimitriou, Philip S. Yu, and Christos Faloutsos. Proximity tracking on time-evolving bipartite graphs. In *SDM*, pages 704–715, 2008.

[22] Hanghang Tong, B. Aditya Prakash, Tina Eliassi-Rad, Michalis Faloutsos, and Christos Faloutsos. Gelling, and melting, large graphs by edge manipulation. In *CIKM*, pages 245–254, 2012.

[23] Konstantin Tretyakov, Abel Armas-Cervantes, Luciano García-Bañuelos, Jaak Vilo, and Marlon Dumas. Fast fully dynamic landmark-based estimation of shortest path distances in very large graphs. In *CIKM*, pages 1785–1794, 2011.

[24] Vijay V. Vazirani. *Approximation algorithms.* Springer-Verlag New York, Inc., New York, NY, USA, 2001.

[25] Xiaohan Zhao, Alessandra Sala, Christo Wilson, Haitao Zheng, and Ben Y. Zhao. Orion: Shortest path estimation for large social graphs. In *WOSN*, 2010.

Query-Based Outlier Detection in Heterogeneous Information Networks

Jonathan Kuck*†, Honglei Zhuang*†, Xifeng Yan‡, Hasan Cam§, and Jiawei Han†
†Department of Computer Science, University of Illinois at Urbana-Champaign
‡Computer Science Department, University of California at Santa Barbara
§US Army Research Lab
{jkuck, hzhuang3, hanj}@illinois.edu, xyan@cs.ucsb.edu, hasan.cam.civ@mail.mil

ABSTRACT

Outlier or anomaly detection in large data sets is a fundamental task in data science, with broad applications. However, in real data sets with high-dimensional space, most outliers are hidden in certain dimensional combinations and are relative to a user's search space and interest. It is often more effective to give power to users and allow them to specify outlier queries flexibly, and the system will then process such mining queries efficiently. In this study, we introduce the concept of query-based outlier in heterogeneous information networks, design a query language to facilitate users to specify such queries flexibly, define a good outlier measure in heterogeneous networks, and study how to process outlier queries efficiently in large data sets. Our experiments on real data sets show that following such a methodology, interesting outliers can be defined and uncovered flexibly and effectively in large heterogeneous networks.

1. INTRODUCTION

Heterogeneous networks are the networks composed of multi-typed, interconnected vertices and links. Since the real world information entities are interconnected, forming numerous, gigantic networks, heterogeneous information networks are ubiquitous and form the basic semantic structure of interconnected data. Thus, detecting anomalies or finding outliers in such networks becomes an important task in network analysis. Although outlier detection has been studied extensively in data mining and various application fields [5, 14], outlier detection in heterogeneous information networks poses several unique challenges:

1. Unlike many outlier analysis methods that work on homogeneous datasets (*e.g.*, find anomalous communications in a communication network), this new endeavor needs fundamental changes on the definition and detection of outliers since it involves heterogeneously typed vertices and links.

2. In a gigantic network, and particularly in a heterogeneous network, it is unrealistic to discover all outliers using "global" techniques. The variety of vertex types, edge types, and paths connecting particular vertices creates many potential viewpoints from which outliers may be classified. These views are difficult to compare and possibly conflicting.

3. Data analysts (as users) need to obtain results promptly to react to outliers or further elaborate their queries. This creates a big challenge to efficiently process outlier queries in heterogeneous information networks.

Based on the above observations, we propose a new outlier detection task, called *query-based outlier detection in heterogeneous information networks*, by facilitating users to compose various kinds of outlier queries flexibly in heterogeneous networks via a novel query language; defining a new outlier measure, called NetOut, to measure the outlierness in such heterogeneous networks; and developing an efficient network detection algorithm for this task. The following is an example that illustrates our ideas.

Motivating example. The DBLP network is a network generated from the computer science bibliographic publication database[1] that consists of a set of vertex types: paper (P), venue (V), author (A), and term (T). A research publication entry essentially generates a set of links of the types $P-V$, $P-A$, and $P-T$, each connecting in the network a paper with its publication venue, set of authors and set of terms, respectively.

It is unrealistic and meaningless to find outliers with respect to all types of the vertices in the entire heterogeneous network. However, it is more interesting to give users freedom to specify what they want. For example, a user can confine the outliers to be among the coauthors of Christos Faloutsos (*i.e.*, all the authors connected with Christos via at least one joint paper).

Even for this author set, it is still unclear what aspect the outliers should be judged by: should the outliers be judged based on their publication venues or their collaborators? The former may lead to finding those who publish multiple papers in rather different venues than the majority of Christos' coauthors; whereas the latter may find those who have rather different collaboration behavior than the majority of his coauthors. Different judgment criteria lead to rather different results, which makes it essential to

*The first two authors made equal contributions.

[1]`http://www.informatik.uni-trier.de/~ley/db`

ask users to specify the criteria explicitly. Furthermore, a user may like to find outliers among Christos' coauthors, not compared within this coauthor set itself but compared with another explicitly specified set, such as prolific EDBT authors (*e.g.*, those who have published at least 10 papers in EDBT).

From this example, one can see that it is necessary to provide an outlier query language with which a user can specify the candidate set (*e.g.*, Christos' coauthors), the aspect (*e.g.*, publishing venues) by which the outliers will be judged, and sometimes the reference set (*e.g.*, prolific EDBT authors). With such primitives, a user can flexibly and unambiguously specify the outliers to be mined in a heterogeneous network.

Besides providing flexible ways for users to interact with the system to specify outlier queries in networks, another important issue is how to define the outlier measures for heterogeneous networks. Taking the query, "finding outlying co-authors of Christos in terms of their publishing venues", it is important to work out a good outlier measure in heterogeneous networks so that one can readily identify the outliers among Christos' co-authors who published multiple papers at rather different venues than that by the majority coauthors of Christos. Such intuition may help us work out a new definition of outlier measure in heterogeneous networks.

In this study, we work on this interesting problem and have made the following contributions.

1. We introduce the concept of query-based outlier in heterogeneous information networks, formalize different components for an outlier query in such networks, and develop a user-friendly, meta-path based outlier query language that allows users to interact with the outlier detection system using their intuition;
2. We introduce a new outlier measure, NetOut, which defines query-based outlierness in heterogeneous information networks, with respect to the queries specified by users;
3. We develop an efficient computation method to find query-based outliers in heterogeneous information networks and analyze our performance improvement in the efficiency study.

The rest of the paper is organized as follows. Section 2 discusses the related work. Section 3 introduces the basic concepts. Section 4 introduces the formal definition of outlier query, and designs a query language as an interface for users to specify outlier queries. Section 5 develops a new outlier measure, NetOut, and shows its effectiveness. Section 6 outlines the implementation of the proposed outlier detection system as well as several query optimization techniques. The performance study of the comparative methods, as well as the efficiency study are reported in Section 7. We present an overall discussion in Section 8 and provide concluding remarks in Section 9.

2. RELATED WORK

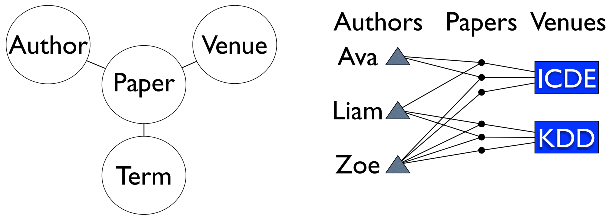
Outlier detection. The field of outlier detection has been explored for years. A good overview of outlier detection techniques can be found in surveys [5, 14].

Our work is most related to the thread of research on networked data outlier detection. There are some early explorations on network outlier detection on bipartite graphs [23]. But most existing studies are confined to homogeneous networks. For example, Gao *et al.* [6] studied contextual outliers in (homogeneous) networks, *i.e.* outliers deviating from their closely connected peers; Akoglu *et al.* [1] proposed OddBall, which takes several network structural properties as features to identify outliers in weighted graphs; Gupta *et al.* [8] studied outliers in terms of their abnormal dynamics among communities; Perozzi *et al.* [19] and Li *et al.* [18] explored outlier detection in attributed graphs. Zong *et al.* [30] studied how to detect abnormal network events and their possible sources. However, these methods are not applicable to heterogeneous information networks. For heterogeneous networks, Gupta *et al.* [7] proposed to measure outlierness based on community distribution of each vertex in the network; Gupta *et al.* [9] also studied outlier detection based on assumption of association-based cliques in networks.

Although a great variety of research has been done on outlier detection given a data set, few of them really consider doing outlier detection in a query-based fashion. Gupta *et al.* [11] proposed using a template subgraph as a query for finding outlier subgraphs, but the definition of a query in this work is not general enough to be extended to most common use cases. Efficient mining of top-k outliers in large databases has been studied in early research. For example, Ramaswamy *et al.* [20] proposed a partition-based algorithm to mine top-k outliers in very large databases, using a distance-based outlier detection algorithm [17]; Jin *et al.* [15] presented an algorithm based on "micro-clusters" to find top-k outliers using the local outlier factor (LOF) measure [4]. Nevertheless, they only optimize for a certain type of outlier definition on the entire data set. Schubert *et al.* [22] proposed a generalized point of view for local outlier detection, but it does not explicitly consider the query-based scenario. In this work we generalize the outlier detection framework and give users flexibility to specify their own definition of an outlier.

Query languages for heterogeneous networks. Managing data organized in heterogeneous information networks is a challenging problem. Compared to a traditional relational database, data is organized in an arbitrary and potentially more complicated graphical structure. There are several different threads of research on developing graph query languages and optimizing queries. A comparison of different graph database models can be found in [2, 3]. Research related to the semantic web usually organizes information into machine-readable web information represented in a language called Resource Description Framework (RDF) [12]. Optimization of the RDF query language SPARQL is studied in [21]. Cypher[2] is another query language utilized by open-source graph database Neo4j; while GraphQL [13] is another query language which supports querying by graph structure. For graph query optimization, Yan *et al.* [26, 27] and Zhao *et al.* [29] proposed indexing strategies to efficiently process graph queries. As knowledge graphs have attracted more studies in recent years, there is also some recent research on querying schema-less graphs. Kasneci *et al.* [16] studies keyword search on knowledge graphs; Yang *et al.* [28] propose an SQL framework where users do not need to specify the querying graph schema/structure precisely. Gupta *et al.* [10] proposed a method to efficiently find the top-k most inter-

[2]`http://docs.neo4j.org/chunked/stable/cypher-introduction.html`



(a) Schema (b) Instantiated network

Figure 1: Bibliographic network schema and instantiated network.

esting subgraphs in a heterogeneous network. However, as far as we know, none of the current graph query languages explicitly support queries for outlier detection in graphs.

3. PRELIMINARIES

Real-world informational or abstract entities are often interconnected, forming multiple gigantic networks. When such networks can be structured around a small number of entity/link types, many interesting properties can be explored systematically. Here we introduce a few related concepts.

Definition 1 (Heterogeneous information network). *A heterogeneous information network [25] is an information network with multiple types of vertices. Without loss of generality, it can be defined as a directed network $G = (\mathcal{V}, \mathcal{E}; \phi, \mathcal{T})$ where $\mathcal{V}$ is the set of vertices, and $\mathcal{E}$ is the set of edges. There is a vertex type mapping function $\phi : \mathcal{V} \to \mathcal{T}$ where $\mathcal{T}$ is the set of types, i.e., each vertex $v \in \mathcal{V}$ belongs to a particular type $T \in \mathcal{T}$. For undirected cases, an undirected edge can be viewed as two symmetric directed edges. When there exists only one vertex type, the network reduces to a homogeneous information network.*

A bibliographic network, such as DBLP, is a heterogeneous information network where there are four types of vertices: paper (P), venue (V), author (A), and term (T), and an edge type represents a type of link between two vertex types (*e.g.*, $P-V$ represents that paper P was published in venue V).

To formalize the relationships between two vertices in a heterogeneous network, meta-paths [24] have been used to represent semantic links at the schema level.

Definition 2 (Meta-path). *In a heterogeneous network G, a meta-path is an ordered sequence of vertex types, denoted as $\mathcal{P} = (T_0 - T_1 - \ldots - T_l)$, or $\mathcal{P} = (T_0T_1 \ldots T_l)$, where $T_x \in \mathcal{T}$.*

In a bibliographic network, $\mathcal{P}_{ca} = (APA)$ is a meta-path, representing the coauthorship between two authors.

We also introduce two basic operators for meta-paths.

Definition 3 (Reversal of a meta-path). *A meta-path $\mathcal{P} = (T_0T_1 \ldots T_l)$ can be reversed, where the reversed path is denoted as $\mathcal{P}^{-1} = (T_lT_{l-1} \ldots T_0)$.*

As an example, if $\mathcal{P} = (APV)$, then its reversal $\mathcal{P}^{-1} = (VPA)$.

Definition 4 (Concatenation of meta-paths). *Given two meta-paths $\mathcal{P}_1 = (T_{1,0} \ldots T_{1,l})$ and $\mathcal{P}_2 = (T_{2,0} \ldots T_{2,l'})$. If $T_{1,l} = T_{2,0}$, then $\mathcal{P}_1$ can be concatenated by $\mathcal{P}_2$, where the concatenated meta-path is denoted as $(\mathcal{P}_1\mathcal{P}_2) = (T_{1,0} \ldots T_{1,l}T_{2,1} \ldots T_{2,l'})$.*

For example, if we have two meta-paths $\mathcal{P}_1 = (APV)$ and $\mathcal{P}_2 = (VPT)$, $\mathcal{P}_1$ can be concatenated by $\mathcal{P}_2$, and the concatenated meta-path $(\mathcal{P}_1\mathcal{P}_2) = (APVPT)$.

Meta-paths provide the schema to instantiate actual paths in a heterogeneous network.

Definition 5 (Meta-path instantiation). *We say an instantiation of $\mathcal{P}$ is a path in G, denoted as $p = (v_0v_1 \ldots v_l)$, satisfying $\phi(v_x) = T_x, \forall x = 0, 1, \ldots, l$. A meta-path can be instantiated by different paths. We represent the set of all path instances of meta-path $\mathcal{P}$ between vertices v_i and v_j by $\pi_{\mathcal{P}}(v_i, v_j)$.*

As an example, for authors Ava and Liam in Figure 1(b), the number of instantiations of meta-path $\mathcal{P}_{ca} = (APA)$ connecting them represents the number of papers they have coauthored, denoted as $|\pi_{\mathcal{P}_{ca}}(\text{Ava, Liam})| = 1$. Similarly, for authors Liam and Zoe, the number of instantiations of meta-path $\mathcal{P}_{ca}$ connecting them is $|\pi_{\mathcal{P}_{ca}}(\text{Liam, Zoe})| = 2$.

Based on the definition of a meta-path, we can define the "neighbors" of a vertex in a heterogeneous network. Different from traditional homogeneous networks, immediate neighbors of a certain vertex could be of different types and therefore have different semantics. Also, vertices that are multiple hops away from the given vertex can be meaningful "neighbors". For the sake of generality, we define the neighborhood of a vertex with respect to a given meta-path. Formally,

Definition 6 (Neighborhood). *In a heterogeneous network G, we define the neighborhood of a certain vertex v_i with regard to a given meta-path $\mathcal{P}$ as $N_{\mathcal{P}}(v_i) = \{v_j | \pi_{\mathcal{P}}(v_i, v_j) \neq \emptyset\}$.*

For example, the set of coauthors of author Zoe in Figure 1(b) can be represented by $N_{\mathcal{P}_{ca}}(\text{Zoe}) = \{\text{Ava, Liam}\}$.

Every vertex v_j, in the neighborhood of vertex v_i, is connected to v_i by at least one instantiation of the specified meta-path. However, multiple instantiations may exist. To better characterize the neighborhood of a vertex, we further define a vector representation of the neighborhood as a "neighbor vector".

Definition 7 (Neighbor vector). *We define a function $\sigma_{\mathcal{P}} : V \mapsto \mathbb{N}^{|V|}$ as the neighbor vector function. With regard to a given meta-path $\mathcal{P}$, it returns the neighbor vector given a certain vertex as input, where the j-th dimension is the count of paths instantiated by $\mathcal{P}$ between v_i and v_j. More precisely,*

$$\sigma_{\mathcal{P}}(v_i) = \Big[|\pi_{\mathcal{P}}(v_i, v_1)|, \ldots, |\pi_{\mathcal{P}}(v_i, v_n)|\Big]$$

For example, given meta-path $\mathcal{P}_{ca}$, Zoe's neighbor vector contains the count of papers co-authored with each of her coauthors, $\sigma_{\mathcal{P}_{ca}}(\text{Zoe}) = [\text{Ava} : 1, \text{Liam} : 2, \text{Zoe} : 5]$. Alternatively, Zoe's neighbor vector given meta-path $\mathcal{P}_v = (APV)$ is the count of papers that she has published in each venue, $\sigma_{\mathcal{P}_v}(\text{Zoe}) = [\text{ICDE} : 2, \text{KDD} : 3]$.

4. OUTLIER QUERIES

In this section, we formalize the definition of a query in the context of outlier detection in heterogeneous information networks. We also design a query language for users to specify queries.

4.1 General Formalization

Generally, a declarative query for outliers consists of two parts, a candidate set containing all the candidates that are potentially meaningful outliers, and a reference set providing a reference for outliers to be compared. In most outlier detection frameworks, the candidate set and the reference set are both assumed to be the entire data set. In our query-based outlier detection framework, users are provided with the flexibility to specify the candidate and reference sets of their interest, which enables our framework to be applicable to a broader range of scenarios.

Another important part of a user query is how the vertices should be compared. In a heterogeneous network, vertices can be compared in many different ways. For example, a pair of authors in a bibliographic information network can be compared based on how much their coauthors overlap, or how many common conferences they attend. Users should be given the flexibility to determine how they would like to compare two vertices.

There are two alternative ways to formulate the comparison method in a query. One is to directly ask the user to define a comparison function $\kappa : S_c \times S_r \mapsto \mathbb{R}$ to compare vertices in the candidate and reference sets; another is to ask the user to declare how a vertex should be characterized, and leave the implementation of the comparison method to the system. In most cases users are clear about the semantics of the desired outliers (*e.g.* comparing two authors based on their coauthors), but do not necessarily understand how to formulate a comparison function accordingly (*e.g.* comparing two authors by the number of their common coauthors), so we adopt the second query formulation where users specify how to characterize vertices using a meta-path based language.

Based on the principles above, we assemble a query for outlier detection with the following modules: the candidate set, the reference set, feature meta-path(s) to specify how a vertex is characterized in the context of outlier detection, and an optional vector used to weight feature meta-paths. To be precise,

DEFINITION 8 (GENERAL OUTLIER QUERY). *An outlier query in a heterogeneous network G is denoted by $Q = (S_c, S_r, \mathbf{P}, \mathbf{w})$, where $S_c \subset V$ is the candidate set of vertices, from which the outliers will be chosen; $S_r \subset V$ is the reference set of vertices, serving as the reference of normal vertices; $\mathbf{P} = (\mathcal{P}_1, \dots, \mathcal{P}_m)$ is a collection of feature meta-paths, describing the user's intuition of which aspects should characterize candidate vertices; $\mathbf{w} \in \mathbb{R}^m$ is a weighting vector for feature meta-paths, and by default is an all-one vector if not specified by users. The outlier detection algorithm should return outliers as a subset of the candidate set, i.e. $\Omega \subset S_c$, that are significantly different from vertices in S_r, in terms of the given meta-paths and weighting vector.*

As an example, if we want to find outliers among Christos Faloutsos' coauthors, then S_c should be defined as all of Christos' coauthors. In the most intuitive scenario the reference set S_r will be the same as S_c. A more complicated query could consist of finding outliers among Christos Faloutsos' coauthors who are unusual with respect to all computer science authors. In this case S_c should still be all of Christos' coauthors, but S_r should be all authors in computer science.

We also need to explicitly state how we are going to define outliers. For example, if we want to find outliers among Christos' coauthors who publish papers in substantially different venues, then it would be appropriate to define a single feature meta-path (APV) to extract all the publishing venues of each author.

Notice that although not explicitly pointed out in the definition, in this paper, we are assuming that all the vertices in $S_c \bigcup S_r$ are of the same type, which is a more intuitive scenario. Also, we require all the meta-paths $\mathcal{P}_1, \dots, \mathcal{P}_m$ has their first element in the same vertex type as vertices in S_c and S_r, otherwise we cannot extract meaningful features from the given feature meta-paths.

To bring this general framework to real-world use cases in heterogeneous information networks, we need a powerful query language for users to specify the query.

4.2 Outlier Query Language

In this subsection, we present an outlier query language. It is capable of effectively supporting most outlier queries in a heterogeneous information network. Outlier detection is not part of the basic functionality supported by traditional SQL languages and relational databases. Due to the complexity of heterogeneous information networks, writing in SQL to specify an outlier detection query can be extremely awkward. Therefore, we define a query language for our outlier detection problem. Notice that although our query language is designed specifically for outlier detection queries in heterogeneous information networks, with minor modification it can also be applied to other types of data sets such as relational databases.

General Formulation. The general structure of a statement for an outlier query is:

```
FIND OUTLIERS FROM ... //Candidate set
COMPARED TO ... //Reference set
JUDGED BY ...  //Feature meta-paths
TOP ...; //Number of outliers to return
```

In the `FROM` or `COMPARED TO` clauses, users can specify a set of vertices. For the `FROM` clause, users specify the candidate set S_c, namely the set of vertices from which outliers are selected. The `COMPARED TO` clause is used to specify the reference set S_r, namely the set of vertices used as a reference. Notice that the `COMPARED TO` clause is optional. If it is not specified, the reference set S_r will be the same as the candidate set S_c.

In the `JUDGED BY` clause, users are required to give a single feature meta-path $\mathcal{P}$ or a collection of feature meta-paths $\mathbf{P}$. The weights of different feature meta-paths may optionally be provided. Vertices in S_r and S_c are compared based on the feature meta-paths and their weights. The top-k outliers, where k is specified in the `TOP` clause, are returned as results.

In the next part of this subsection, we introduce how to actually specify a set of vertices, and how to specify a collections of (weighted) feature meta-paths. Then we give several examples.

Specifying candidate/reference set. In the simplest case, users can refer to a certain vertex by its type and name:

```
venue{"EDBT"}
```

which returns all the venue-typed vertices with exactly the name "EDBT".

In many cases, users are interested in outliers in a certain local area in the network. Therefore, we allow the user to specify the neighborhood of a certain vertex, with regard to the definition in Section 3. Recall that to define a neighborhood requires a specific vertex v_i and meta-path $\mathcal{P}$. We use the dot operator to concatenate different types and represent a meta-path, where the first element is a specified vertex. As an example:

```
venue{"EDBT"}.paper.author
```

returns the neighborhood of venue-typed vertex EDBT with respect to meta-path (VPA). More formally it returns $N_{\mathcal{P}}(v_i)$, where v_i is the vertex that represents the venue EDBT and $\mathcal{P} = (VPA)$. Semantically, it is the set of all the authors with papers published in the venue EDBT.

We also allow users to specify additional conditions in a `WHERE` clause, to further restrict the vertices selected in the candidate or reference set. For example, the set of authors who have published in the conference EDBT and who have published more than 10 papers can be specified as:

```
venue{"EDBT"}.paper.author AS A
WHERE COUNT(A.paper) > 10
```

Multiple SQL-style operations can be applied to extract each vertex set. For instance, a user can generate the union of multiple sets using the `UNION` operator:

```
venue{"EDBT"}.paper.author
UNION
venue{"ICDE"}.paper.author
```

which will return the set of authors who have published in EDBT or ICDE.

Alternatively, a user can generate the intersection of several sets using the `INTERSECT` operator:

```
venue{"EDBT"}.paper.author
INTERSECT
venue{"ICDE"}.paper.author
```

which will return the set of authors who have published in both EDBT and ICDE.

Specifying feature meta-paths. A meta-path in our query language can simply be represented as an ordered list of types separated by dots. For example, in a bibliographic network, we can specify the feature meta-path (APV) to compare authors with respect to their publishing venues as:

```
author.paper.venue
```

in the `JUDGED BY` clause.

If there are multiple aspects, namely multiple feature meta-paths, that a user would like to use when classifying outliers, we separate different meta-paths by commas "`,`". For example, a user may judge outlier authors based on both their publishing venues and their coauthors. Meta-paths (APV) and (APA) are given as feature meta-paths and we write in the `JUDGED BY` clause:

```
author.paper.venue, author.paper.author
```

We have shown that our query language can support the specification of a collection of feature meta-paths $\mathbf{P}$. When users want to define different weights for different meta-paths, we also allow users to specify the weights in this query language, by writing the weight after a colon following the corresponding meta-path. As an example, suppose the user wants to judge outliers based on both their publishing venues and their coauthors, weighting the venues with twice the importance of coauthors. We can write in the `JUDGED BY` clause

```
author.paper.venue: 2.0, author.paper.author
```

where as `author.paper.author` is explicitly assigned a weight, it is by default weighted as 1.

Notice, we require that all specified feature meta-paths have the same type in their first element as vertices in S_c and S_r.

4.3 Example queries

Example 1. To find the top-10 outliers among Christos' coauthors in terms of venues they publish (*i.e.*, find 10 authors in Christos' coauthor who publish in the weirdest venues), we write the query:

```
FIND OUTLIERS
FROM author{"Christos Faloutsos"}.paper.author
JUDGED BY author.paper.venue
TOP 10;
```

Since no reference set is specified, the outliers are determined by comparing with others in the candidate set A.

Example 2. Alternatively, a user might want to find outliers in Christos' coauthors who are significantly different from the authors in the KDD community, in terms of the venues they publish in and their coauthors. We can write this query as:

```
FIND OUTLIERS
FROM
  author{"Christos Faloutsos"}.paper.author
COMPARED TO
  venue{"KDD"}.paper.author
JUDGED BY
  author.paper.venue,
  author.paper.author
TOP 10;
```

Example 3. To find the top-50 outliers among SIGMOD authors, who have published at least 5 papers, with respect to their coauthors (weight 1) and the vocabulary used in their paper titles (weight 3), we write the query:

```
FIND OUTLIERS
FROM venue{"SIGMOD"}.paper.author AS A
     WHERE COUNT(A.paper) >= 5
JUDGED BY
     author.paper.author,
     author.paper.term : 3.0
TOP 50;
```

5. NETWORK-BASED OUTLIER MEASURE: NETOUT

There have been many outlierness measures for numerical and categorical data. However, defining a good outlierness measure for use in heterogeneous information networks is still a challenging problem. The major challenge is the ambiguity of outlier semantics, as there are multiple types of paths connecting vertices.

Basic principle. In this section we define the properties of an outlier in a heterogeneous information network given a specific query. We address the problem for the query of finding outlier vertices among a set of candidate vertices with respect to a set of reference vertices, when judged by a specific feature meta-path. The feature meta-path and sets of candidate and reference vertices are given in the query formulation. The definition should be intuitive while utilizing the rich information provided by the network.

In general, an outlier among a group is an object that differs substantially from the rest of the group. In the context of finding outliers in a network, we look for vertices that are least connected to the group, but it is also important to consider each vertex's maximum potential for connectivity when comparing its group connectivity with that of other vertices. In the context of our specific problem, we follow the basic principle that outlying vertices should be most structurally disconnected from the reference set, with respect to their expected potential for connectivity.

5.1 Normalized Connectivity

We begin by presenting a measure to express the connectivity between two individual vertices with respect to their potential connectivity. Later we will apply it between individual candidate vertices and all vertices in the reference set to determine an outlier score for each candidate vertex.

In our query language we allow the user to specify a collection of feature meta-paths $\mathbf{P}$. In this section we only consider queries where $\mathbf{P}$ consists of a single feature meta-path $\mathcal{P}$. Finding outliers given a collection of weighted feature meta-paths can be done in a number of ways. The connectivity between vertices can be redefined, or independent outlier scores can be computed considering each feature meta-path independently and then averaged. We leave the problem of determining the best method to a future study.

The meta-path $\mathcal{P}$ can be viewed by the user as specifying a traditional feature type which will be used to judge the outlierness of each candidate vertex. We are interested in finding outliers that are most structurally disconnected, so we construct the symmetric meta-path linking the candidate type to itself, $\mathcal{P}_{sym} = (\mathcal{P}\mathcal{P}^{-1})$.

We can now define the connectivity κ between two vertices, v_a and v_b, as the number of path instantiations of $\mathcal{P}_{sym}$ between the two vertices, $\kappa(v_a, v_b) = |\pi_{\mathcal{P}_{sym}}(v_a, v_b)|$. The visibility of vertex v_a is the connectivity between v_a and itself, $\kappa(v_a, v_a)$, which is a measure of a vertex's potential connectivity with other vertices. We define the normalized connectivity between vertices v_a and v_b as the ratio of their connectivity to v_a's visibility:

Definition 9 (Normalized Connectivity). *Given heterogeneous network G containing two vertices v_a and v_b of type $T \in \mathcal{T}$ and symmetric meta-path $\mathcal{P}_{sym} = (\mathcal{P}\mathcal{P}^{-1}) = (T \dots T)$, the normalized connectivity between v_a and v_b is defined as* $\tilde{\kappa}(v_a, v_b) = \frac{\kappa(v_a,v_b)}{\kappa(v_a,v_a)} = \frac{|\pi_{\mathcal{P}_{sym}}(v_a,v_b)|}{|\pi_{\mathcal{P}_{sym}}(v_a,v_a)|}$

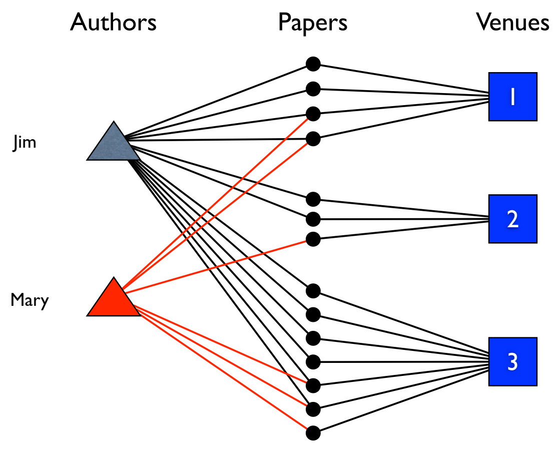


Figure 2: Path instantiations of the meta-path $(APVPA)$ connecting authors Jim and Mary

Note that $\tilde{\kappa}(v_a, v_b) \neq \tilde{\kappa}(v_b, v_a)$ when $\kappa(v_a, v_a) \neq \kappa(v_b, v_b)$. Normalized connectivity can be interpreted in terms of a random walk beginning at v_a along meta-path $\mathcal{P}_{sym}$. The probability of ending at v_b is $\frac{|\pi_{\mathcal{P}_{sym}}(v_a,v_b)|}{||\sigma_{\mathcal{P}_{sym}}(v_i)||_1}$, where σ is the neigbor vector function defined in Section 3. The probability of returning to v_a is $\frac{|\pi_{\mathcal{P}_{sym}}(v_a,v_a)|}{||\sigma_{\mathcal{P}_{sym}}(v_i)||_1}$. The probability of ending at v_b divided by the probability of returning to v_a is then $\frac{|\pi_{\mathcal{P}}(v_a,v_b)|}{|\pi_{\mathcal{P}}(v_a,v_a)|}$, which is the normalized connectivity $\tilde{\kappa}(v_a, v_b)$. This fits with our intuition well. The probability of returning to v_a acts as a normalization constant such that the normalized connectivity between v_a and itself will always be 1. When v_a is more connected to v_b than itself, $\tilde{\kappa}(v_a, v_b) > 1$. When v_a is less connected to v_b than itself, $\tilde{\kappa}(v_a, v_b) < 1$. Comparing $\tilde{\kappa}(v_a, v_c)$ with $\tilde{\kappa}(v_b, v_c)$ shows whether it is more likely to arrive at v_c in a random walk beginning at v_a or v_b (normalized by the likelihood of returning to the original vertex).

Example 4. We use a concrete example (Cf. Figure 2) to illustrate the behavior of normalized connectivity. We examine two authors Jim and Mary in a bibliographic network G given feature meta-path $\mathcal{P} = (APV)$.

The connectivity (path count) between Jim and Mary is $2\times4+1\times2+3\times6 = 28$. The normalized connectivities in this example are $\tilde{\kappa}(Jim, Mary) = 0.5$ and $\tilde{\kappa}(Mary, Jim) = 2$. This reflects that Jim's connectivity with Mary is half his connectivity with himself, while Mary's connectivity with Jim is twice that with herself.

5.2 Outlier Measure: NetOut

To measure a certain vertex v_i's outlierness with regard to a given reference set S_r we sum the normalized connectivity between v_i and all vertices in S_r. This gives v_i's connectivity to the reference set as a whole, normalized by its potential connectivity. The lower this normalized group connectivity, the more likely that v_i is an outlier. We define the outlierness measure NetOut as:

Definition 10 (Outlierness in Heterogeneous Networks: NetOut). *In a heterogeneous network G, given a query Q, for any $v_i \in S_c$, the outlierness can be measured by*

$$\Omega_{NetOut}(v_i;Q) = \sum_{v_j \in S_r} \tilde{\kappa}(v_i, v_j)$$

where smaller Ω values correspond to greater likelihood of being an outlier. We refer to $\Omega_{NetOut}(v_i;Q)$ as simply $\Omega(v_i;Q)$ outside this section when there is no potential ambiguity.

Rather than summing v_i's normalized connectivity with every vertex in the reference set, we could find the minimum or maximum normalized connectivity between v_i and any vertex in the reference set. In many cases finding the minimum normalized connectivity is not meaningful because many vertices in the candidate set are completely disconnected from at least one vertex in the reference set. To evaluate the usefulness of finding the maximum normalized connectivity consider two vertices v_i and v_j. Vertex v_i is moderately connected to one vertex in the reference set but completely disconnected from every other context vertex. Vertex v_j is weakly connected to every vertex in the reference set. In most cases it is hard to justify that v_j should be a stronger outlier than v_i.

Summing the normalized connectivities has the additional advantage of computational efficiency. Computing NetOut for every vertex in the candidate set can be reduced to an $\mathcal{O}(|S_r|+|S_c|)$ operation. In comparison, using the minimum or maximum normalized connectivity instead would always require $\mathcal{O}(|S_r| \times |S_c|)$ time.

Next we justify our use of normalized connectivity when defining NetOut by comparing with the similarity measures PathSim and cosine similarity.

PathSim. Superficially it may appear that a similarity measure could be used instead of normalized connectivity in our outlier detection problem. The normalized connectivity between two vertices is not a true similarity measure because it lacks symmetry. In this section we introduce the similarity measure PathSim for comparison, to justify the need for normalized connectivity.

In a previous study of similarity search in heterogenous information networks [24], PathSim was introduced as an interesting measure to define network-based structural similarity. The PathSim measure between two vertices v_i and v_j following a meta-path $\mathcal{P}$ in a heterogeneous information network is defined as,

$$PathSim_{\mathcal{P}_{sym}}(v_i, v_j) = \frac{|\pi_{\mathcal{P}_{sym}}(v_i, v_j)|}{\left(|\pi_{\mathcal{P}_{sym}}(v_i, v_i)| + |\pi_{\mathcal{P}_{sym}}(v_j, v_j)|\right)/2}$$

For comparison purposes we define:

$$\Omega_{PathSim}(v_i;Q) = \sum_{v_j \in S_r} PathSim_{\mathcal{P}_{sym}}(v_i, v_j)$$

$PathSim_{\mathcal{P}_{sym}}(v_i, v_j)$ is defined by the connectivity between v_i and v_j divided by the average of v_i and v_j's visibility. Based on this formula, PathSim assigns high similarity values to the vertices that are strongly connected (*i.e.*, there are many paths between v_i and v_j following the meta-path) but having low average visibility (*i.e.*, there are not many other paths from v_i or v_j reaching v_i or v_j itself).

PathSim has demonstrated its promise at similarity search in heterogeneous information networks. Comparing to SimRank or Personalized PageRank, PathSim assigns lower similarity to vertices whose connectivity is high but whose visibilities differ.

Cosine Similarity. We define a comparable version of NetOut using the cosine similarity instead of normalized connectivity:

$$\Omega_{CosSim}(v_i;Q) = \sum_{v_j \in S_r} \frac{\sigma_{\mathcal{P}}(v_i) \cdot \sigma_{\mathcal{P}}(v_j)}{||\sigma_{\mathcal{P}}(v_i)||_2 \times ||\sigma_{\mathcal{P}}(v_j)||_2}$$

Where $\sigma_{\mathcal{P}}(v_i)$ is the neighbor vector function defined in Section 3.

NetOut Example. We consider a toy example to demonstrate NetOut's properties. Table 1 shows the publication records of candidate authors. In this example we consider a query giving the reference set composed of 100 authors with publication records identical to Sarah's and feature meta-path $\mathcal{P} = (APV)$. Table 2 shows NetOut scores for each candidate author. We compare with outlier scores computed using PathSim and the cosine similarity in place of normalized connectivity in the NetOut formula.

Sarah is clearly not an outlier, with $\Omega(Sarah;Q) = 100$ (normalized connectivity with identical vertices of 1 multiplied by the size of the reference set). Rob has an unusual publication record and a low NetOut score, signifying he is a potential outlier. Lucy's publication differs from authors in the reference set, but is more similar than Robs, giving her a higher NetOut score than Rob.

Next we compare NetOut scores with outlier scores computed using PathSim and the cosine similarity in place of normalized connectivity. PathSim is computed using the meta-path $(APVPA)$. The cosine similarity is computed using each author's neighbor vector (defined in Section 3). All three measures find the same outlier ordering for Sarah, Rob, and Lucy.

Joe has published only two papers in the venue SIGGRAPH. While SIGGRAPH is an unusual venue, Joe's publication record is currently unstable and likely to change over the course of his life. It is possible that his first publications are simply noise. NetOut does not classify Joe as an outlier. While his connectivity with authors in the reference set is low, this is expected because of his low visibility. In a random walk beginning at Joe following the meta-path $(APVPA)$, the probability of reaching an author in the reference set is the same as the probability of returning to Joe. However, the PathSim and cosine similarity versions both classify Joe as an outlier with very low scores.

Emma is clearly a very unusual author, and this is apparent in her NetOut score. She has only published in the unusual venue SIGGRAPH and she has published more papers than the authors in the reference set, so we can assume her publication record is stable at this point. Her outlier score computed using PathSim is actually higher than Joe's, because her visibility is more similar to the visibility of authors in the reference set. Emma's outlier score computed using the cosine similarity is the same as Joe's. Both have neighbor vectors with the same direction, so their cosine similarity with other authors is identical. NetOut computed using nor-

Table 1: Publication records of candidate and reference vertices. The reference set contains 100 authors with identical publication records, given by the reference author.

	VLDB	KDD	STOC	SIGGRAPH
Reference Author	10	10	1	1
Sarah	10	10	1	1
Rob	0	1	20	20
Lucy	0	5	10	10
Joe	0	0	0	2
Emma	0	0	0	30

Table 2: NetOut outlier scores of select candidate vertices given a query whose feature meta-path is $\mathcal{P} = (APV)$ and reference set is given in Table 1, compared with scores computed using PathSim and the cosine similarity in place of normalized connectivity.

	Ω_{NetOut}	$\Omega_{PathSim}$	Ω_{CosSim}
Sarah	100	100	100
Rob	6.24	9.97	12.43
Lucy	31.11	32.79	32.83
Joe	50	1.94	7.04
Emma	3.33	5.44	7.04

malized connectivity finds outliers without bias towards any particular visibility, while PathSim and the cosine similarity are biased towards authors with low visibility.

NetOut Experimental Comparison. To further explain our use of normalized connectivity, rather than a symmetric similarity measure such as PathSim or CosSim, we employ a concrete example on DBLP data set to compare the results returned by different methods.

We construct a query, to find top-5 outliers among all the coauthors of Christos Faloutsos, in terms of their publishing venues. The context and candidate sets are specified as Faloutsos' co-authors and the feature meta-path is given by $\mathcal{P} = (APV)$.

The comparison results are shown in Table 3. The top outliers found by NetOut defined using normalized connectivity are active in fields besides data mining, which is Christos' primary focus, and have a wide range of visibilities. Adam Wright has published roughly 30 papers, while Katia P. Sycara has published roughly 300 papers. In contrast, all the top-5 outliers found by PathSim or CosSim are authors who have published less than 2 papers, which makes them uninteresting as outliers. This further demonstrates the inherent bias towards candidate vertices with low visibility when using PathSim or the cosine similarity.

6. IMPLEMENTATION

In this section we briefly introduce some technical details regarding the implementation of our query-based outlier detection system. We first introduce a basic baseline implementation and then optimizations to improve the efficiency of query execution.

6.1 Baseline

There are two basic steps to execute an outlier query: retrieve the candidate and reference sets S_c and S_r and calculate the outlierness of each vertex in the reference set based on the given feature meta-paths.

For retrieval of S_c and S_r, the basic operations are finding a vertex v_i given its name and type and then traversing the network from v_i while counting the instantiations of the given meta-path. The first operation can be naively implemented by a hash table, or a trie, which is relatively efficient. The second basic operation, materializing a meta-path $\mathcal{P}$, has time complexity exponential to the length of $\mathcal{P}$.

A naïve way to calculate the outlierness measure would be to first calculate the normalized connectivity $\tilde{\kappa}(\cdot,\cdot)$ between each vertex in the candidate set S_c and each vertex in the references set S_r, then sum up all the $\tilde{\kappa}(v_i,\cdot)$ for each vertex in S_c to obtain the outlierness. However, this has time complexity of $O(|S_r| \times |S_c|)$.

Recall the definition of connectivity, $\kappa(\cdot,\cdot)$:

$$\begin{aligned}\kappa(v_i, v_j) &= |\pi_{\mathcal{P}_{sym}}(v_i, v_j)| \\ &= \sigma_{\mathcal{P}}(v_i) \cdot \sigma_{\mathcal{P}}(v_j)\end{aligned}$$

The calculation of NetOut can be re-written as:

$$\begin{aligned}\Omega(v_i; Q) &= \sum_{v_j \in S_r} \tilde{\kappa}(v_i, v_j) \\ &= \sum_{v_j \in S_r} \frac{\kappa(v_i, v_j)}{\kappa(v_i, v_i)} \\ &= \sum_{v_j \in S_r} \frac{\sigma_{\mathcal{P}}(v_i) \cdot \sigma_{\mathcal{P}}(v_j)}{\sigma_{\mathcal{P}}(v_i) \cdot \sigma_{\mathcal{P}}(v_i)} \\ &= \frac{1}{\|\sigma_{\mathcal{P}}(v_i)\|_2^2} \left(\sigma_{\mathcal{P}}(v_i) \cdot \left(\sum_{v_j \in S_r} \sigma_{\mathcal{P}}(v_j) \right) \right) \quad (1)\end{aligned}$$

Notice that the term $\sum_{v_j \in S_r} \sigma_{\mathcal{P}}(v_j)$ remains the same for all $v_i \in S_c$. Therefore we can first calculate it, then calculate the outlierness value NetOut for all vertices $v_i \in S_c$. Therefore, the time complexity of calculating NetOut for every candidate vertex is only $O(|S_r| + |S_c|)$.

However, even if the calculation of NetOut is efficient, it is still relatively slow compared to actually obtaining the neighbor vector $\sigma_{\mathcal{P}}(v_i)$ for a given v_i and meta-path $\mathcal{P}$. Materializing this neighbor vector requires traversal of the heterogeneous network, which can be time-consuming when the specified meta-path is long or the degree of the vertex of interest is high. Therefore we aim to optimize the query processing time by reducing the materialization time of meta-paths.

6.2 Optimization

Pre-materialization. To accelerate the materialization of meta-paths, we can pre-compute the materialization of length-2 meta-paths. Depending on the pattern of user queries we may compute all length-2 paths or only a subset. To be more precise, for each vertex $v_i \in V$, and all possible $\mathcal{P}$ such that $|\mathcal{P}| = 2$, we can calculate and store the vector $\sigma_P(v_i)$.

In the execution of a query, it may be necessary to calculate $\sigma_P(v_i)$ for an arbitrary meta-path $\mathcal{P}$. We can always decompose $\mathcal{P}$ into several length-2 meta-paths as

Table 3: Comparing different outlierness measure, with query $S_c = S_r$ =author("Christos Faloutsos").paper.author and feature meta-path $\mathcal{P} = (APV)$ Outliers found by normalized connectivity are interesting outliers, while outliers found by PathSim or CosSim are authors with very few papers, which are not interesting.

Method	Ω_{NetOut}		$\Omega_{PathSim}$		Ω_{CosSim}	
Ranking	Name	Ω-value	Name	Ω-value	Name	Ω-value
1	Adam Wright	2.54	Wenyao Ho	1.07	John Chien-Han Tseng	0.0022
2	Philip Koopman	2.55	Fernanda Balem	1.12	Fernanda Balem	0.0038
3	Nicholas D. Sidiropoulos	3.29	Rebecca B. Buchheit	1.31	Guoqiang Shan	0.0046
4	Katia P. Sycara	3.64	John Chien-Han Tseng	1.41	Wenyao Ho	0.0066
5	David S. Doermann	3.65	Chi-Dong Chen	1.47	Chi-Dong Chen	0.0077

$\mathcal{P} = (\mathcal{P}_1 \cdots \mathcal{P}_k)$, where $|\mathcal{P}_1| = \cdots = |\mathcal{P}_{k-1}| = 2$. If the original meta-path $\mathcal{P}$ is even-length, then $|\mathcal{P}_k| = 2$.

Notice that for any $\mathcal{P} = (\mathcal{P}_1\mathcal{P}_2)$, we have

$$\begin{aligned}\sigma_{\mathcal{P}}(v_i) &= \sum_{v_j} |\pi_{\mathcal{P}_1}(v_i, v_j)| \sigma_{\mathcal{P}_2}(v_j) \\ &= [\sigma_{\mathcal{P}_2}(v_1), \ldots, \sigma_{\mathcal{P}_2}(v_n)]\, \sigma_{\mathcal{P}_1}(v_i)\end{aligned}$$

which implies that by decomposing an arbitrary meta-path $\mathcal{P}$ into several length-2 meta-paths, we can calculate $\sigma_{\mathcal{P}}(v_i)$ by multiplication of indexed vectors. Even if the original meta-path is odd-length, we only need to traverse the network for a single hop. Retrieving an index can be of $O(1)$ by storing the vectors in a hash table and the time complexity of multiplication is affordable when the vectors are sparse.

By efficiently retrieving $\sigma_{\mathcal{P}}(v_i)$, multiple steps in the query processing benefit, including the retrieval of candidate set S_c and reference set S_r, and the calculation of connectivity functions.

Selective pre-materialization. The aforementioned indexing strategy pre-calculates the indexed vectors for all vertices with regard to all length-2 meta-paths. This exhaustive indexing strategy guarantees efficiency improvement, but can also result in a large index table. To achieve reasonable efficiency while conserving memory, we may only want to construct length-2 meta-paths starting from a certain set of vertices.

To this end, a strategy is to count the frequency with which different vertices appear in queries. The query set used for selecting vertices for building indices is referred to as "initialization query set" for SPM. The initialization query set can be existing query logs, or else synthetic queries when query logs are not available. A certain absolute or relative threshold is set, and length-2 meta-paths are only computed beginning at vertices that appear in queries with frequency above the set threshold.

7. EXPERIMENTAL RESULTS

In this section we evaluate experimental performance.

7.1 Experiment Setup

Data set. We employ a bibliographic data set from ArnetMiner[3] to construct a heterogeneous information network. The data set consists of 2,244,018 publications and 1,274,360 authors in the field of computer science. The heterogeneous network contains 4 types of vertices: paper, venue, author and term. Possible type of edges include paper-author (written-by), paper-venue (published in) and paper-term (title contains).

[3]`http://arnetminer.org/AMinerNetwork`

Table 4: Query templates used to construct query sets for efficiency experiments. 10,000 random authors are selected and substituted where indicated by "·" in each query template.

Number	Query Templates
$\mathcal{Q}_1$	`FIND OUTLIERS FROM author{·}.paper.author` `JUDGED BY author.paper.venue` `TOP 10;`
$\mathcal{Q}_2$	`FIND OUTLIERS IN author{·}.paper.venue` `JUDGED BY venue.paper.term` `TOP 10;`
$\mathcal{Q}_3$	`FIND OUTLIERS IN author{·}.paper.term` `JUDGED BY term.paper.venue` `TOP 10;`

Query sets. In order to check the efficiency performance of our algorithm, we randomly select 10,000 author-typed vertices from the heterogeneous information networks. Three different types of queries are shown in Table 4, which are referred to as "query templates". For each template, we substitute the randomly selected vertices into the position indicated by the dot "·", to generate 10,000 queries. We refer to each set of queries as $\mathcal{Q}_i$. These randomly generated query sets are used in efficiency studies.

Comparison methods. In efficiency studies, we compare the following implementations.

- *Baseline.* The baseline implementation without pre-materialization (Cf. Equation (1)).
- *Pre-Materialization (PM).* All length-2 meta-path instantiations are pre-computed and stored.
- *Selective Pre-Materialization (SPM).* A subset of all length-2 meta-path instantiations are pre-computed and stored, for selected vertices that frequently appear in S_c given a set of specified queries, where the relative frequency threshold is set to 0.01.

We use the set of all possible queries for the given query template as the initialization query set in SPM.

7.2 Case Study

We examine the effectiveness of our proposed outlierness measure by checking the experimental results of several typical queries. The results are summarized in Table 5.

In our first two experiments we use Christos Faloutsos' coauthors as the candidate and reference sets. We use `author.paper.venue` in the first experiment as the single feature meta-path and `author.paper.author` in the second.

The first query we try is to find outliers with regard to their publishing venues. The returned top-10 outliers of Christos' coauthors are actually quite deviated from his research field (with one exception), which is data mining. For example, Adam Wright works on biomedical informatics; Philip Koopman is in the area of embedded systems. Interestingly, Nicholas D. Sidiropoulos publishes most of his work in the community of signal processing. However, one of his research interests is tensor analytics and mining, which is closely related to Christos Faloutsos' research interests. As we are judging outliers based on publishing communities, Nicholas D. Sidiropoulos is still listed as one of the top outliers. Although most of the aforementioned outliers are relatively established authors in their own fields, John Chien-Han Tseng is a student who has published only one paper in the venue KDIR (a very rare venue for authors in the reference set to publish in). Tseng's appearance demonstrates that our method does not discriminate against candidate outliers based on their visibility.

In the second query, we still search for outliers among Christos' coauthors, but judged by their coauthors. The results are substantially different from the first query, with only one overlapping author (Katia P. Sycara). This is evidence that in a heterogeneous information network outliers can be reasonably defined in multiple ways, resulting in totally different outcomes. Without user-specified queries, mining outliers can be an ill-defined problem leading to semantic ambiguity. The top outliers are still mainly in fields other than data mining, with an interesting exception: Ee-Peng Lim is a researcher who also focuses on social network analysis and mining [4], with a significant number of papers published in data mining venues. Lim is still listed as an outlier among Christos' coauthors, as his collaborator network does not overlap much with Christos' collaborators. This is a typical example of the importance of providing a specific outlier definition. Outliers under one definition could be totally normal given another definition.

In the third query, we attempt to find outliers among KDD authors, with respect to their publishing venues. The top outlier turns out to be "NULL" which represents missing data. Other top outliers are also interesting: Wolfgang Glänzel is a professor of economics and business, with the majority of his papers published in economic related venues; Paul M. Thompson has published most papers in medical or neuroscience venues.

7.3 Efficiency Studies

We also examine the efficiency performance of our different query optimization strategies. In this experiment, we process the query sets generated from the query template in Table 4 and measure the system performance by query processing time.

[4]`https://sites.google.com/site/aseplim/`

Table 5: Case study of NetOut results on several queries.

$S_c = S_r =$ author("Christos Faloutsos").paper.author
$\mathcal{P} =$ author.paper.venue

Ranking	Name	Ω-value
1	Adam Wright	2.54
2	Philip Koopman	2.55
3	Nicholas D. Sidiropoulos	3.29
4	Katia P. Sycara	3.64
5	David S. Doermann	3.65
6	Asim Smailagic	3.69
7	John Chien-Han Tseng	4.00
8	Daniel P. Siewiorek	4.22
9	Jessica K. Hodgins	4.52
10	Dimitris N. Metaxas	4.57

$S_c = S_r =$ author("Christos Faloutsos").paper.author
$\mathcal{P} =$ author.paper.author

Ranking	Name	Ω-value
1	Dimitris N. Metaxas	1.06
2	Bin Zhang	1.06
3	Hui Zhang	1.07
4	Lionel M. Ni	1.07
5	Bin Liu	1.08
6	Joel H. Saltz	1.08
7	Yang Wang	1.08
8	Hao Wang	1.08
9	Ee-Peng Lim	1.12
10	Katia P. Sycara	1.13

$S_c = S_r =$ venue("KDD").paper.author
$\mathcal{P} =$ author.paper.venue

1	*NULL*	1.27
2	Wolfgang Glänzel	4.99
3	Paul M. Thompson	6.46
4	Yehuda Lindell	9.21
5	Kwan-Liu Ma	12.2
6	Dhabaleswar K. Panda	13.23
7	Christos Davatzikos	13.95
8	Andrzej Skowron	14.62
9	Anil K. Jain	15.75
10	Fillia Makedon	15.95

Improved efficiency with pre-materialization. In Figure 3 we compare the performance of the baseline implementation, the implementation with all length-2 meta-paths pre-materialized (PM), and the selective pre-materialized version with relative frequency threshold 0.01 (SPM). With pre-materialization the efficiency can always be improved significantly, 5-100 times faster than the baseline implementation. This verifies the effectiveness of the indexing strategy. The performance of SPM is generally worse than the fully materialized version PM, but is more than 10 times faster than the baseline in query set $\mathcal{Q}_3$.

In-depth efficiency analysis of SPM. For the SPM strategy, we conduct a study to look into the processing time spent on different parts. As shown in Figure 4, For almost all query sets, most of the processing time is spent on materializing feature meta-paths of vertices without pre-materialization. Loading pre-stored instantiations of feature meta-paths for vertices with materialization is the least time

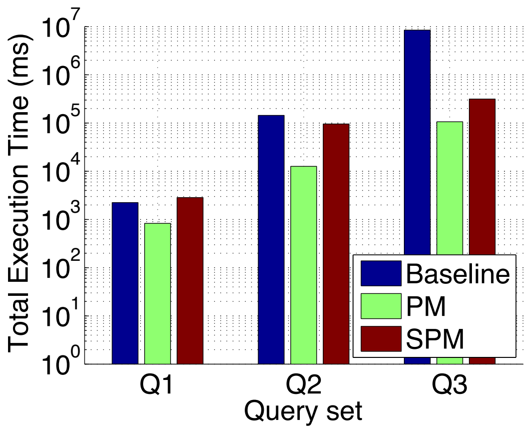


Figure 3: Comparing total execution time for 10,000 randomly generated queries between the baseline implementation and the implementation with pre-materialization.

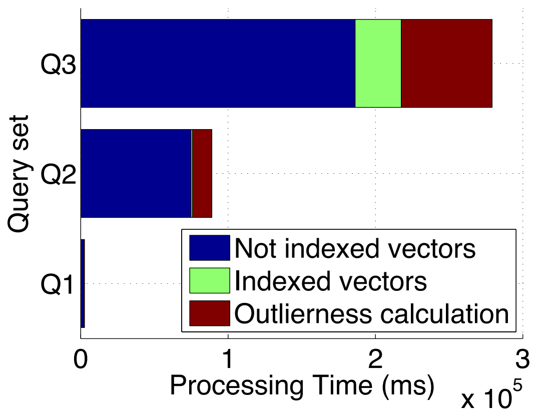
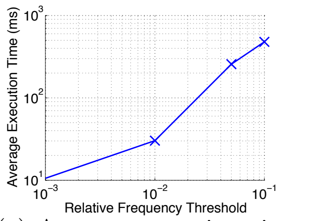


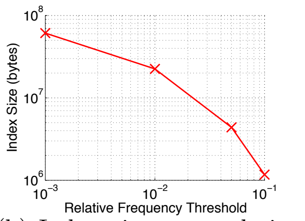
Figure 4: In-depth analysis of query processing time using selective pre-materialization strategies with the relative frequency threshold set to 0.01. "Not indexed vectors" indicates processing time spent on meta-path materialization from vertices without pre-materialization; "Indexed vectors" indicates time spent looking up pre-materialized meta-paths from materialized vertices; "Outlierness calculation" indicates calculation time of NetOut.

consuming part, while calculating NetOut can be slower. Calculating inner products between vectors is potentially more expensive than retrieving vectors from indices.

Threshold studies for SPM. We check the performance of SPM strategies with different relative frequency threshold. We construct indices with the relative frequency threshold set at 0.001, 0.01, 0.05, and 0.1 respectively, and compare both the processing time and index size, as shown in Figure 5. Not surprisingly, the index size decreases as the threshold rises, while the average query processing time also increases. A relatively optimal threshold is likely to be found between 0.01 and 0.05, considering both factors.



(a) Average execution time vs. relative frequency threshold



(b) Index size vs. relative frequency threshold

Figure 5: Comparison of efficiency performance with different relative frequency threshold in selective pre-materialization indexing strategy.

8. DISCUSSION

Alternative query language design. There are other ways to define the query language with more generality. It is possible to allow users to specify functions that are not meta-path based for measuring the similarity between two vertices, or to allow users to define their own outlierness measure, *etc.*. However, maximizing the generality will require users to have more expertise knowledge, which violates our principle to provide users with a more declarative language. In comparison, our language design is simple and satisfies most needs for data analysis.

Outlierness measure. The outlierness measure NetOut we defined in this paper is easy to compute compared to many state of the art outlier detection algorithms. It is still possible to substitute other outlier detection algorithms based on our query-based outlier detection framework, as long as they support the input specified by our queries. However, most of them are not efficient enough to be suited for users' exploratory query behavior. Our experiments comparing with other outlier detection algorithms (*e.g.* LOF [4]) suggest that they cannont produce better results than NetOut.

Extensions. Although we frame our query-based outlier detection study in a closed-schema heterogeneous information network data set, our framework can easily be extended to a broader range of data sets. For example, our query language can be applied to open-schema networks such as a knowledge graph, and the baseline implementation of NetOut should also be applicable. It is also possible to apply our query-based outlier detection idea on traditional relational databases, with a structure similar to our defined outlier query language, but changing the meta-path-based language into SQL. It would be interesting to develop a query-based outlier detection system for different types of data sets, based on our defined framework and query language, while exploring the implementation challenges.

There are additional directions to further facilitate users' exploratory interaction with the system. For example, instead of returning the top-k outliers after the user specifies the query, it might be helpful to visualize outliers to provide more insight. Alternatively, the system could find the approximate top-k outliers, with confidences, while the query is being processed so that users can determine whether to continue processing the query. The system might even be able to suggest how the users can modify their queries to get more interesting, or more unusual, outliers.

9. CONCLUSION

In this paper, we propose a query-based outlier detection framework. We design a query language for outlier detection in heterogeneous information networks, which gives users flexibility to mine various types of outliers based on their intuition. We also propose NetOut, a novel meta-path based outlierness measure for mining outliers in heterogeneous networks, and show its effectiveness compared to other outlierness measures. We finally present implementation details, where we utilize pre-materialization and selective pre-materialization to optimize query processing time. Experimental results show that our proposed query-based outlier detection framework can efficiently return meaningful results for a range of queries.

Acknowledgements. Research was sponsored in part by the Army Research Lab. under Cooperative Agreement No. W911NF-09-2-0053 (NSCTA) and W911NF-11-2-0086, the Army Research Office under Cooperative Agreement No. W911NF-13-1-0193, National Science Foundation IIS-1017362, IIS-1320617, and IIS-1354329, HDTRA1-10-1-0120, NIH Big Data to Knowledge (BD2K) (U54), and MIAS, a DHS-IDS Center for Multimodal Information Access and Synthesis at UIUC.

10. REFERENCES

[1] L. Akoglu, M. McGlohon, and C. Faloutsos. Oddball: Spotting anomalies in weighted graphs. In *PAKDD*, pages 410–421. Springer, 2010.

[2] R. Angles. A comparison of current graph database models. In *ICDE Workshops*, pages 171–177. IEEE, 2012.

[3] R. Angles and C. Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1, 2008.

[4] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander. Lof: identifying density-based local outliers. In *SIGMOD*, pages 93–104. ACM, 2000.

[5] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Computing Surveys*, 41(3):15:1–15:58, 2009.

[6] J. Gao, F. Liang, W. Fan, C. Wang, Y. Sun, and J. Han. On community outliers and their efficient detection in information networks. In *KDD*, pages 813–822. ACM, 2010.

[7] M. Gupta, J. Gao, and J. Han. Community distribution outlier detection in heterogeneous information networks. In *ECML/PKDD*, pages 557–573. Springer, 2013.

[8] M. Gupta, J. Gao, Y. Sun, and J. Han. Integrating community matching and outlier detection for mining evolutionary community outliers. In *KDD*, pages 859–867. ACM, 2012.

[9] M. Gupta, J. Gao, X. Yan, H. Cam, and J. Han. On detecting association-based clique outliers in heterogeneous information networks. In *ASONAM*, pages 108–115. IEEE, 2013.

[10] M. Gupta, J. Gao, X. Yan, H. Cam, and J. Han. Top-k interesting subgraph discovery in information networks. In *ICDE*, pages 820–831. IEEE, 2014.

[11] M. Gupta, A. Mallya, S. Roy, J. H. Cho, and J. Han. Local learning for mining outlier subgraphs from network datasets. In *SDM*, 2014.

[12] C. Gutierrez, C. Hurtado, and A. O. Mendelzon. Foundations of semantic web databases. In *PODS*, pages 95–106. ACM, 2004.

[13] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *SIGMOD*, pages 405–418. ACM, 2008.

[14] V. J. Hodge and J. Austin. A survey of outlier detection methodologies. *Artificial Intelligence Review*, 22(2):85–126, 2004.

[15] W. Jin, A. K. Tung, and J. Han. Mining top-n local outliers in large databases. In *KDD*, pages 293–298. ACM, 2001.

[16] G. Kasneci, F. M. Suchanek, G. Ifrim, M. Ramanath, and G. Weikum. Naga: Searching and ranking knowledge. In *ICDE*, pages 953–962. IEEE, 2008.

[17] E. M. Knox and R. T. Ng. Algorithms for mining distancebased outliers in large datasets. In *VLDB*, pages 392–403, 1998.

[18] N. Li, H. Sun, K. Chipman, J. George, and X. Yan. A probabilistic approach to uncovering attributed graph anomalies. In *SDM*, 2014.

[19] B. Perozzi, L. Akoglu, P. Iglesias Sánchez, and E. Müller. Focused clustering and outlier detection in large attributed graphs. In *KDD*, pages 1346–1355. ACM, 2014.

[20] S. Ramaswamy, R. Rastogi, and K. Shim. Efficient algorithms for mining outliers from large data sets. In *SIGMOD*, volume 29, pages 427–438. ACM, 2000.

[21] M. Schmidt, M. Meier, and G. Lausen. Foundations of sparql query optimization. In *ICDT*, pages 4–33. ACM, 2010.

[22] E. Schubert, A. Zimek, and H.-P. Kriegel. Local outlier detection reconsidered: a generalized view on locality with applications to spatial, video, and network outlier detection. *Data Mining and Knowledge Discovery*, 28(1):190–237, 2014.

[23] J. Sun, H. Qu, D. Chakrabarti, and C. Faloutsos. Neighborhood formation and anomaly detection in bipartite graphs. In *ICDM*, pages 418–425, 2005.

[24] Y. Sun, J. Han, X. Yan, P. S. Yu, and T. Wu. Pathsim: Meta path-based top-k similarity search in heterogeneous information networks. *Proceedings of the VLDB Endowment*, 4(11), 2011.

[25] Y. Sun, B. Norick, J. Han, X. Yan, P. S. Yu, and X. Yu. Integrating meta-path selection with user-guided object clustering in heterogeneous information networks. In *KDD*, pages 1348–1356. ACM, 2012.

[26] X. Yan, P. S. Yu, and J. Han. Graph indexing: a frequent structure-based approach. In *SIGMOD*, pages 335–346. ACM, 2004.

[27] X. Yan, P. S. Yu, and J. Han. Substructure similarity search in graph databases. In *SIGMOD*, 2005.

[28] S. Yang, Y. Wu, H. Sun, and X. Yan. Schemaless and structureless graph querying. *Proceedings of the VLDB Endowment*, 7(7), 2014.

[29] P. Zhao and J. Han. On graph query optimization in large networks. *Proceedings of the VLDB Endowment*, 3(1-2):340–351, 2010.

[30] B. Zong, Y. Wu, J. Song, A. K. Singh, H. Cam, J. Han, and X. Yan. Towards scalable critical alert mining. In *KDD*, pages 1057–1066. ACM, 2014.

Efficient caching for constrained skyline queries

Michael Lind Mortensen [#1], Sean Chester [#2], Ira Assent [#3], Matteo Magnani [*4]

[#] *Aarhus University, Denmark* [*] *Uppsala University, Sweden*

[1] illio@cs.au.dk [2] schester@cs.au.dk [3] ira@cs.au.dk [4] matteo.magnani@it.uu.se

ABSTRACT

Constrained skyline queries retrieve all points that optimize some user's preferences subject to orthogonal range constraints, but at significant computational cost. This paper is the first to propose caching to improve constrained skyline query response time. Because arbitrary range constraints are unlikely to match a cached query exactly, our proposed method identifies and exploits similar cached queries to reduce the computational overhead of subsequent ones.

We consider interactive users posing a string of similar queries and show how these can be classified into four cases based on how they overlap cached queries. For each we present a specialized solution. For the general case of independent users, we introduce the Missing Points Region (MPR), that minimizes disk reads, and an approximation of the MPR. An extensive experimental evaluation reveals that the querying for an (approximate) MPR drastically reduces both fetch times and skyline computation.

1. INTRODUCTION

Constrainted Skyline A constrained skyline query [3] is an effective way of filtering a constrained dataset to points that express all optimal trade-offs of the dataset's attributes. For example, if searching for cheap 3+ star hotels near a conference venue, one hotel is said to *dominate* another if it is at least as highly rated, well priced and near as the latter, yet strictly better than the other hotel on at least one of these metrics. The *constrained skyline* is the set of points that satisfy the given constraints and are not dominated by any others that satisfy the constraints. In practice, the constraints are critical in allowing users to determine the skyline on the data relevant to them. E.g. average income earners may not be interested in luxury hotels nor backpacker hostels, so a non-constrained skyline cannot capture their preferences. Constraints reduce the input size, yet, paradoxically, makes computing the skyline quite challenging, because, unlike an unconstrained skyline which can simply be pre-materialized, the skyline points are unpredictable.

The naive approach, presented in [3], is to execute a range query to fetch points satisfying the constraints, and then compute the skyline over those points using an efficient skyline algorithm (e.g., [7, 8, 16, 23]). This has the advantage of simplicity, but the performance is highly sensitive to the selectivity of the range query. The best known technique is the I/O-optimal *BBS* algorithm [19], which uses an R-tree index and a heap-based priority queue to guide the search for skyline points, while pruning paths in an R-Tree if outside the constraints. In this paper, we outperform *BBS* by reusing partial query solutions.

Caching A fair assumption is that many users will pose constrained skyline queries on the same dataset. Where users have similar needs, the constraint regions likely overlap. For example, young backpackers will all typically search with price constraints that match a cheap budget, producing similar queries. Business travellers, conversely, may be more concerned with location than price; a distinct set of similar queries.

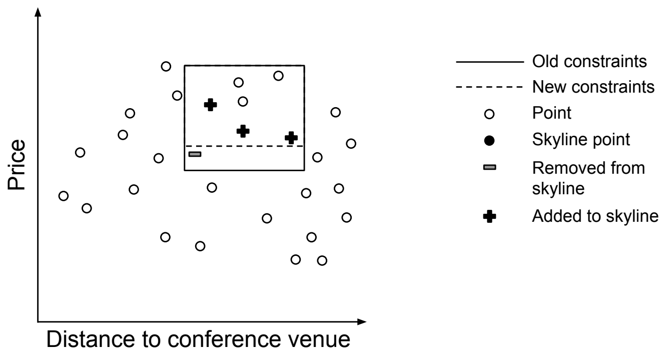


Figure 1: Small constraint changes have large impacts on skylines

Additionally, an iterative, exploratory query-refine cycle is common in search tasks [18], where a user issues a query, observes the results, and then adjusts constraints to manipulate the results. Hence, even a single user can produce strings of highly similar queries, each with distinct skylines [6, 17].

So, this paper addresses a natural question: *how can the results of a constrained skyline query be reused to speed up subsequent, similar ones?* In contrast to existing techniques (e.g., [3, 19]), we can obtain significant speed-up by decomposing a range query into disjoint smaller ones, and discarding those that a previous cached query result implies are unnecessary, hence reading fewer data points than the isolated query necessitates.

Challenges Despite the apparent simplicity of overlapping two range queries to compare results, caching constrained skylines is deceptively challenging. Unlike previous research on caching (subspace) skyline queries [2, 14, 20] where constraints are not considered, cache hits with exact matching constraints are quite unlikely, especially for real-valued and high-dimensional data. Therefore, we investigate how to infer partial skyline solutions from *overlapping*, rather than matching, query constraints, which would yield a low cache hit rate.

This, however, introduces a new challenge, because small changes to constraints can have a profound impact on the skyline [6]. For example, Figure 1 shows an "old" (solid) and "new" (dashed) constrained skyline query on a toy hotel example, where the minor increase to the $Price$-constraint from the "old" to the "new" query is enough to eliminate a skyline point (minus), which promotes three previously dominated points into the skyline (plus). To address this challenge, we introduce the notion of *stability* which characterizes when solution points will be shared among old and new queries. Even in difficult, *unstable* cases, we show how a previous query's solution points, even when not satisfying the current constraints, can prune the current search range.

Finally, dimensionality poses a natural challenge for caching constrained skylines by increasing the pruning complexity (as we show in Section 5.3). Therefore, we present an effective approximation technique that balances the number and selectivity of small range queries. As a result, we can outperform baseline and the I/O-optimal BBS algorithms by several factors, and scale elegantly with increasing workloads.

Contributions Despite the cost of constrained skylines, this paper is the first to investigate how caching can drastically improve their running time. After reviewing related work (Section 2) we introduce the problem formally (Section 3) and make the following novel contributions (Sections 4-6) before concluding the paper (Section 8).

- For the exploratory use case, in which subsequent queries differ only by one constraint, we present a case-by-case breakdown of the four possible overlap relationships, along with how to compute the constrained skyline for each (Section 4);
- For the general case of arbitrary constraint overlap, we introduce an algorithm to compute the *Missing Points Region* (MPR), which is the minimal region that must be queried from disk. We also introduce an *Approximate MPR* (aMPR), sacrificing minimality for fewer independent range queries (Section 5);
- We introduce a caching algorithm, *Cache-Based Constrained Skyline* (CBCS) that handles cache searching, management and use based on the earlier analysis (Section 6); and
- We conduct an extensive experimental evaluation of our method to show when our caching yields superior efficiency for related queries relative to baselines and state-of-the-art (Section 7).

2. RELATED WORK

Constrained skylines The skyline operator [3] was introduced along with a straightforward extension to constrained skyline queries that first retrieves all data satisfying the constraints, and then applies any skyline algorithm (e.g., [7,8,16,23]). Subsequently, the R-tree-based method *BBS* [19] supports constraints by pruning paths in the R-tree if they are outside the constraint region. *BBS* is I/O-optimal and state-of-the-art when not using caching. We include it in our empirical study (Sect. 7).

For arbitrary subsets of dimensions, known as subspaces, [10] partitions data and queries vertically onto several low-dimensional R-trees. Without subspaces, their approach is essentially a constraint-based version of the *NN* method [15], shown in [19] to be inferior to *BBS* for constrained skylines. [9] study distributed constrained skylines where distributed local skylines are merged into a global result. Efficiency gains come from computing independent local skylines at data sites in parallel, meaning that a non-distributed application is equivalent to computing the constrained skyline naively. [1] study constrained subspace skylines in a horizontally partitioned P2P environment. Constrained subspace skylines are computed in order of potential dominance on each node, avoiding those pruned by earlier nodes. It suffers the same limitations as [9]. [22] study continuous constrained skyline queries for streams, determining areas that could influence the current skyline. The problem is different from ours, namely maintenance of fixed constrained skylines for dynamic data, rather than dynamic constraints. [11] study query optimization of *Semi-skylines*, which use partial order preferences. If applied only to traditional constraints, the method corresponds to recomputation from scratch for any change in constraints. [4] estimates the cardinality of (constrained) skylines in a DBMS and can be used to assess which skyline algorithm to apply in the naive approach.

A user study [17] uses existing constrained skyline algorithms to investigate how users understand, issue and interact with constrained skylines. Finally, [6] study how dynamic changes of constraints and subspaces affect skyline membership. Neither [6] nor [17] offer algorithmic contributions.

Caching skylines [12] and [5] study caching of subspace skylines in a P2P setting using local caches with a superpeer network and a centralized index, respectively. Neither support constrained skylines. [2] study caching of subspace skylines, where results are cached directly and used to answer queries in related subspaces. [14] caches partial-order domain user preferences to process queries with similar user preferences. [20] caches dynamic skylines, where domination is based on the distance to a query. None of them consider constraints and thus suffer from the same issues as [12] and [5].

In conclusion, existing constrained skylines algorithms demand recomputation from scratch if constraints differ even slightly. Also, existing caching approaches only support identical constraints, which is unlikely to occur in practice, especially when considering, e.g., exploratory search scenarios, real-valued data, multiple users and several dimensions. In this work, we limit the number of points read and dominance tests performed, by reusing cached results on similar constraints.

3. PRELIMINARIES

Let S be a set of data points over an ordered set of numerical dimensions D, where the value of $s \in S$ in dimension $i \in D$ is denoted $s[i]$. A *set of constraints*, $C = \langle \underline{C}, \overline{C} \rangle$, is a pair of points indicating the minimum value, $\underline{C}[i]$, and the maximum value, $\overline{C}[i]$, for each dimension $i \in D$. A *constraint region*, R_C, is the set of all possible points satisfying constraints C:

$$R_C = \left\{ p \in R^{|D|} \mid \forall i \in D : \underline{C}[i] \leq p[i] \leq \overline{C}[i] \right\}.$$

Observe that R_C describes a $|D|$-dimensional hyper-rectangle, like the rectangles in Figure 1. Similarly, the *constrained data*, S_C, is the set of data points that satisfy constraints C:

$$S_C = \left\{ s \in S \mid \forall i \in D : \underline{C}[i] \leq s[i] \leq \overline{C}[i] \right\}.$$

We note the following properties relating constraint regions and constrained data:

1. Given constraints C, the set of points S_C satisfying constraints C form a (possibly empty) subset of the set of points in the region R_C described by C: $S_C \subseteq R_C$.

Data & space notation	
S	Dataset S
D	Dimensions of S
$R = \left\{p \in \mathbb{R}^{\lvert D \rvert}\right\}$	Region of potential $\lvert D \rvert$-dimensional points
S_C	Points in S limited by constraints C
R_C	Region R limited by constraints C
p, q	Points in region R
s, u, t, v	Points in dataset S.
$p[i], s[i]$	Value in dimension i of point p, s, resp.
Query notation	
$C = \langle \underline{C}, \overline{C} \rangle$	Constraints consisting of low/high limits
$\underline{C}[i], \overline{C}[i]$	Lower/upper constraint on dimension i
$Sky(S, C)$	Skyline on S constr. by C
$s \succ t$	Point s dominates point t
$DR(s)$	Dominance region of point s
$DR(s, C)$	Dominance region of point s constrained by C
$RQ(C) = (S_C \cap R_C)$	Range query on the region constrained by C

Table 1: Notation

2. (Data determines space) $s \in S_C \implies s \in R_C$.

3. (Space only contains constrained data) $p \in R_C \implies p \in S_C \vee p \notin S$.

In this paper we study how to efficiently answer *constrained skyline queries* (Def. 1) given S and C, if an in-memory cache (Def. 3) is available. The skyline is defined through the concept of *dominance* [3]: a point $s \in S$ (or, analogously, $s \in R$) *dominates* another point $t \in S$, denoted $s \succ t$, iff $\exists i \in D$: $s[i] < t[i]$ and $\forall j \in D, s[j] \leq t[j]$.[1] In order words, s is at least as small as t on every attribute, and strictly smaller on at least one.

The *conventional skyline* of S, denoted $Sky(S)$, is the subset of points not dominated by any other points in S:

$$Sky(S) = \{s \in S \mid \nexists t \in S : t \succ s\}.$$

These are exclusively those points that minimize some linear function over the dataset's dimensions. Figure 2 illustrates the skyline with black points for our running hotel example.

The *constrained skyline* is the set of points that satisfy the constraints while not being dominated by any other points that also satisfy the constraints (Definition 1). Equivalently, it is the skyline over input S_C.

Definition 1 (Constrained skyline [3, 19])**.**
Given S, C, the constrained skyline*, denoted $Sky(S, C)$, is:*

$$Sky(S, C) = Sky(S_C) = \{s \in S_C \mid \nexists t \in S_C : t \succ s\}.$$

Figure 2 also illustrates a *constrained skyline* as the set of gray points in the rectangle spanned by constraints $C = \langle \underline{C}, \overline{C} \rangle$. Note here again that the constrained skyline can be very different from the conventional skyline (black points).

Every point $s \in S$ (or $s \in R$) has a *dominance region* [10], denoted $DR(s)$, which is the hyper-rectangular region in which any point p is dominated by s (Definition 2).

[1]Assumed without loss of generality: a preference for maximization can be handled by multiplying an attribute by -1.

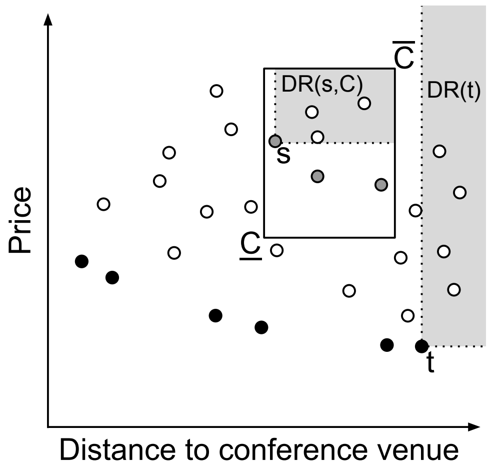


Figure 2: Illustration of a skyline (black points) and constrained skyline (grey points inside the rectangle) on our running example. Also shown are dominance regions (solid gray rectangles).

Definition 2 (Dominance region [10])**.**
For point $s \in S$, the dominance region *is defined as:*

$$DR(s) = \{p \in R \mid s \succ p\}.$$

For any $s \in S$, $DR(s) \cap Sky(S) = \emptyset$; so dominance regions help detect subsets of points that need not be fetched from disk. In the presence of constraints C, each point s also induces a *constrained dominance region*, denoted $DR(s, C)$, which is the portion of $DR(s)$ that satisfies C. The gray rectangles in Figure 2 illustrate $DR(t)$ for a conventional skyline point and $DR(s, C)$ for a constrained skyline point.

Lastly, our objective is to resolve constrained skyline queries using in-memory constrained skyline cache items. A *cache item* (Definition 3) is a 3-tuple consisting of an earlier queried set of constraints, the resultant constrained skyline, and the skyline's *minimum bounding rectangle* (*MBR*).

Definition 3 (Constrained skyline cache)**.**
An in-memory cache holding n cache items *$\{\mathcal{I}_1, \ldots, \mathcal{I}_n\}$, where each cache item $\mathcal{I}_i$ is a 3-tuple:*

$$\mathcal{I}_i = \langle Sky(S, C), \mathrm{MBR}, C \rangle$$

$Sky(S, C)$ is the skyline result on constraints C and MBR *is the minimum bounding rectangle of $Sky(S, C)$.*

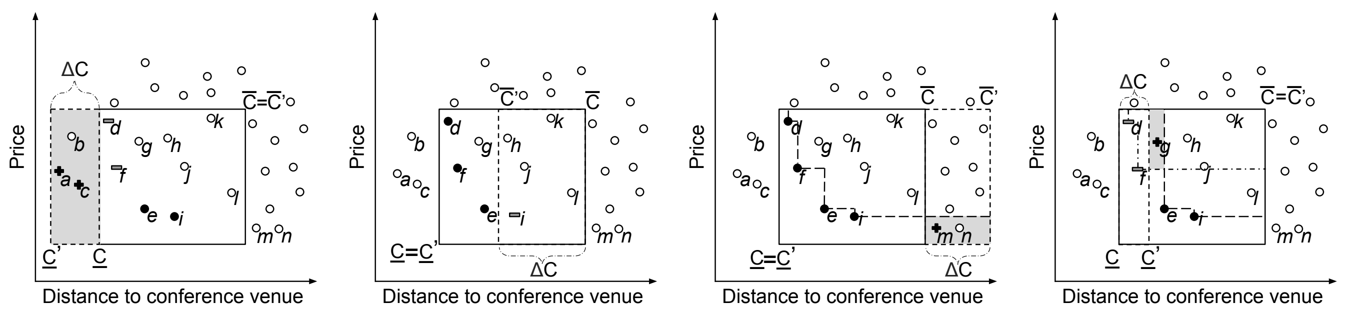
With the notation in place (and summarized in Table 1), the problem studied in this paper can now be stated as follows:

Problem Statement (Cache-based constrained skyline).
Given S, C', and an in-memory cache $\{\mathcal{I}_1, \ldots, \mathcal{I}_n\}$, utilize a cache item $\mathcal{I}_i$ to compute $Sky(S, C')$ without fetching all of $S_{C'}$.

4. EXPLOITING RELATED QUERIES

Recall from Section 2 that existing caching techniques for skylines require an exact match on constraints. A user who continually modifies, say, the *price* or *distance* constraints as he/she refines his/her hotel search therefore produces a long string of cache misses, despite having made only small, incremental changes to his/her query.

In this section, we focus on these *incremental changes*, where old constraints C and new constraints C' overlap in all but one



(a) Decreasing lower constraint (b) Decreasing upper constraint (c) Increasing upper constraint (d) Increasing lower constraint

Figure 3: Cases (a)-(d) of incrementally changing one constraint at a time; our solutions fetch only the points in the gray regions.

dimension. In doing so, we both address the potential for large computational savings in this principal case and build intuition for the general methods presented in Section 5.

Specifically, we use $Sky(S, C)$ to limit how much of $S_{C'}$ that must be fetched from disk to determine $Sky(S, C')$. We first introduce the concept of *stability* to characterize when constrained skylines share solution points (Section 4.1). Then, we identify the four possible (and easily detectable) manners in which incremental constraint changes may overlap, presenting specialized solutions for each (Section 4.2).

4.1 Skyline stability

Clearly, a cache item for $Sky(S, C)$ indicates which points from S are in $Sky(S, C)$. More importantly, it also implies which points from S_C *are not* in $Sky(S, C)$. *Stability* (Definition 4) captures this insight relative to new constraints, C'.

Definition 4 (Constrained skyline stability)**.**
We say that $Sky(S, C)$ is stable *relative to C' iff:*

$$s \in Sky(S, C') \implies (s \notin S_C) \vee (s \in Sky(S, C))$$

In other words, $Sky(S, C)$ is stable relative to C' if points from S_C not in $Sky(S, C)$ are also not in $Sky(S, C')$. Otherwise, we call $Sky(S, C)$ *unstable* relative to C'. We observe in Theorem 1 that stability is guaranteed when, for all $i \in D$, $\underline{C}[i] \geq \underline{C}'[i]$ (or, trivally, when constraints do not overlap).

Theorem 1 (Guaranteed stability)**.**
$Sky(S, C)$ is guaranteed to be stable relative to C' iff:

$$(\forall i \in D \colon \underline{C}'[i] \leq \underline{C}[i]) \vee$$
$$(\exists i \in D \colon \underline{C}'[i] > \overline{C}[i] \vee \overline{C}'[i] < \underline{C}[i])$$

The full proof of Theorem 1 is in Appendix 9, but the intuition is that new constraints can only invalidate the skyline if they shrink the constraint region, removing skyline points and thereby their dominance region influence on the skyline result. Stability is gauranteed since no point $s \in S_C$ dominated by removed point $t \in Sky(S, C)$ can satisfy an upper constraint that t does not satisfy.

From Definition 4 and Theorem 1 come two natural consequences. First, for stable cases, we need only fetch points in the new part of the constraint region that did not satisfy the old constraints (Corollary 1). Second, a skyline result is unstable if and only if an "old" skyline point $s \in Sky(S, C)$ is outside the "new" constraints C', and it dominated points that still satisfy the new constraints (Corollary 2).

Corollary 1. *If $Sky(S, C)$ is stable relative to C' then:*

$$\forall s \in Sky(S, C') \colon \ (s \in Sky(S, C)) \ \underline{\vee} \ (s \in (S_{C'} \setminus S_C))$$

Corollary 2. *$Sky(S, C)$ is unstable relative to C' iff:*

$$\exists t \in Sky(S, C) \colon t \notin S_{C'} \ \wedge$$
$$\exists s \in (S_C \cap S_{C'}) \colon t \succ s \ \wedge$$
$$\nexists u \in (Sky(S, C) \cap S_{C'}) \colon u \succ s$$

4.2 Incremental constraint changes

With the theory of stability in place, we show how any incremental change can be solved with minimum points read. For each of four possible cases, we prove the correctness and minimality (proofs in Appendix 9) of our solution and illustrate the intuition of the ideas by example/illustration.

Figures 3a-3d show the four cases for incremental changes of constraints C (the solid rectangle), one dimension at a time: (a) decreasing a lower constraint, (b) decreasing an upper constraint, (c) increasing an upper constraint and (d) increasing a lower constraint. Note that we always have only these four cases, regardless of dimensionality. The initial constraints $C = \langle \underline{C}, \overline{C} \rangle$ and new constraints C', as well as the change ΔC from C to C' are displayed in each figure. For each case, the part of $S_{C'}$ that we fetch is enclosed in the gray region. Note that while the illustrated gray regions are all rectangular, this only holds for $|D| = 2$ as we will show in Section 5.3.

Case (a): Decreasing a lower constraint (Fig. 3a) From Theorem 1, $Sky(S, C)$ is stable relative to C', and from Corollary 1 all new skyline points lie in ΔC. Instead of fetching all of $S_{C'}$, we can fetch just the points in $(R_{C'} \setminus R_C)$. Further pruning is not possible: no points in $Sky(S, C)$ dominate any part of ΔC.

Theorem 2 (Case (a) solution)**.**
If $\overline{C}' = \overline{C}$, $\exists i \colon \underline{C}'[i] < \underline{C}[i]$, $\forall j \in D \setminus \{i\} \colon \underline{C}'[j] = \underline{C}[j]$, then:

$$Sky(S, C') = Sky(Sky(S, C) \cup S_{\Delta C}, C')$$

In the example (Fig. 3a), a, b and c are fetched from the database with a, c as new skyline points (illustrated by a plus sign), while existing d, f are dominated by a and c under constraints C' (illustrated by a minus sign). The final skyline is a, c, e, i. Without the cached $Sky(S, C)$, we must read 12 points, a-l, from disk.

Case (b): Decreasing an upper constraint (Fig. 3b) Again, from Theorem 1, $Sky(S, C)$ is stable wrt C'. From Corollary 1 the skyline points in $Sky(S, C')$ are in $Sky(S, C)$ or in ΔC. Since $R_{C'}$ is enclosed in R_C, we need simply remove the previous skyline points not satisfying the new constraints.

Theorem 3 (Case (b) solution).
If $\underline{C}' = \underline{C}$, $\exists i\colon \overline{C}'[i] < \overline{C}[i]$, $\forall j \in D \setminus \{i\}\colon \overline{C}'[j] = \overline{C}[j]$, *then:*

$$Sky(S, C') = Sky(S, C) \cap S_{C'}$$

In this example (Fig. 3b), only i falls outside the new constraints and is simply removed to obtain the new skyline.

Case (c): Increasing an upper constraint (Fig. 3c) As before, $Sky(S, C)$ is stable relative to C' (Theorem 1), and new skyline points in $Sky(S, C')$ are in $Sky(S, C)$ or in ΔC (Cor. 1). Unlike before, however, we use the dominance regions of points in $Sky(S, C')$ to further prune parts of ΔC:

Theorem 4 (Case (c) solution).
If $\underline{C}' = \underline{C}$, $\exists i\colon \overline{C}'[i] > \overline{C}[i]$, $\forall j \in D \setminus \{i\}\colon \overline{C}'[j] = \overline{C}[j]$, *then:*

$$Sky(S, C') = Sky(Sky(S, C) \cup \{s \in S_{\Delta C} \mid \nexists t \in Sky(S, C)\colon t \succ s\}, C')$$

We thus prune $S_{\Delta C}$ such that we only read $((S_{C'} \setminus S_C) \setminus \{s \in S_{\Delta C} \mid \exists t \in Sky(S, C)\colon t \succ s\})$. In the example (Fig. 3c), $R_{C'}$ contains 18 points, the logic of Case (a) reduces it to 9 points, and we eventually fetch only 2 points, m and n.

Case (d): Increasing a lower constraint (Fig. 3d) Unlike Cases (a)-(c), $Sky(S, C)$ is not stable relative to C'. Despite this instability, we can use the old skyline result by determining invalidated parts of the cache item and reevaluate these under constraints C'. Using what is left of $Sky(S, C)$ within the queried constraints C' we prune regions before reading from disk. Since no two skyline points dominate each other (Def. 1), the remaining part of $Sky(S, C)$ is not invalidated:

Theorem 5 (Case (d) solution).
If $\overline{C}' = \overline{C}$, $\exists i\colon \underline{C}'[i] > \underline{C}[i]$, $\forall j \in D \setminus \{i\}\colon \underline{C}'[j] = \underline{C}[j]$, *then:*

$$\begin{aligned} Sky(S, C') = Sky((&Sky(S, C) \cap S_{C'}) \cup \\ &\{s \in (S_C \cap S_{C'}) \mid \\ &\exists t \in (Sky(S, C) \cap S_{\Delta C})\colon t \succ s \ \wedge \\ &\nexists u \in (Sky(S, C) \cap S_{C'})\colon u \succ s\}, C') \end{aligned}$$

Thus, we avoid reading $(S_C \cap S_{C'})$ fully, and retrieve only $((S_C \cap S_{C'}) \setminus \{s \in (S_C \cap S_{C'}) \mid \nexists t \in (Sky(S, C) \cap S_{\Delta C})\colon t \succ s \vee \exists u \in (Sky(S, C) \cap S_{C'})\colon u \succ s\})$.

In the example (Fig. 3d) instead of 7 points, we only fetch g. Throughout all examples, we save the latency of fetching unnecessary points, and the cost of conducting dominance tests over an otherwise larger input.

5. ARBITRARY CONSTRAINT CHANGES

In this section, we build on the intuition from Section 4 to handle the general case where the number of constraint changes is arbitrary. We first generalize the gray regions of the previous section into the *Missing Points Region* (*MPR*), the minimal area that must be fetched (Section 5.1), and introduce an efficient algorithm to compute it (Section 5.2). We then illustrate how the MPR grows arbitrarily complex with dataset dimensionality (Section 5.3) and introduce an effective approximation to reduce that complexity (Section 5.3).

5.1 The Missing Points Region

Given constraints C and C', the *Missing Points Region* (the gray rectangles in Figure 3) is the minimum, possibly disjoint, region of points for which neither C' nor $Sky(S, C)$ can be used to infer said points' inclusion/exclusion in $Sky(S, C')$. It is comprised of those parts of $R_{C'}$ that do not overlap the dominance region of any point $s \in Sky(S, C)$, lies outside R_C, or, in unstable cases, where $\exists t \in (Sky(S, C) \cap (S_C \setminus S_{C'})), s \in (S_C \cap S_{C'})\colon t \succ s$ (Definition 5).

Definition 5. *(Missing Points Region)*
Given $Sky(S, C)$*,* C'*, the* Missing Points Region*,* MPR*, is:*

$$\begin{aligned} \mathrm{MPR} = \{p \in R_{C'} \mid (&p \in (R_{C'} \setminus (R_C \cap R_{C'})) \ \vee \\ &\exists t \in (Sky(S, C) \cap (R_C \setminus R_{C'}))\colon p \in DR(t, C)) \ \wedge \\ &\nexists u \in (Sky(S, C) \cap R_{C'})\colon p \in DR(u, C')\} \end{aligned}$$

The MPR is both *complete* and *minimal* in the sense that, with knowledge only of $Sky(S, C)$ and C', any point in MPR could be in $Sky(S, C')$ (Theorems 6 and 7, respectively).

Theorem 6. *(Completeness)*
Given $Sky(S, C)$*,* C'*, where* $R_C \cap R_{C'} \neq \emptyset$*, we have:*

$$Sky(S, C') = Sky((Sky(S, C) \cap S_{C'}) \cup (\mathrm{MPR} \cap S_{C'}), C')$$

The full proof of Theorem 6 is in Appendix 9, but the intuition is that there are only two ways in which points can be missing: (1) Expansion of C and (2) Invalidation of C. All expanded and invalidated areas are fetched unless guaranteed excluded by known points from $Sky(S, C)$. The remaining non-invalidated regions of S_C remain stable.

Theorem 7. *(Minimality)*
Given only $Sky(S, C)$*,* C'*, where* $S_C \cap S_{C'} \neq \emptyset$*, any point in* $\mathrm{MPR} \cap S_{C'}$ *could be in* $Sky(S, C')$*.*

The full proof of Theorem 7 is also in Appendix 9, but the intuition is that by definition no known point outside MPR can dominate a point inside MPR, only points inside MPR can dominate each other. Thus to minimize MPR further, we must know the contents of MPR, which we cannot do without fetching the points in MPR.

5.2 Computing the MPR

We present our algorithm to compute the MPR, which, per Theorem 7 is used to minimize points fetched. Our general approach is to start with the hyper-rectangle $\langle \underline{C}', \overline{C}' \rangle$ and continually split it using C and $Sky(S, C)$ into sub-hyper-rectangles such that many of them can be immediately discarded. At the end, we are left with a set $\mathcal{H}$ of disjoint, axis-orthogonal hyper-rectangles (i.e., range queries) covering the exact region of the MPR.

The advantage of this approach is three-fold: 1) it calculates the MPR in a form (set of range queries) that can be queried directly; 2) the primary operation, splitting axis-orthogonal hyper-rectangles with axis-orthogonal hyperplanes, is simple and efficient; and 3) the continual discarding of hyper-rectangles controls $|\mathcal{H}|$, important because the algorithm runs $\mathcal{O}(|\mathcal{H}| \cdot |Sky(S, C)| \cdot |D|)$.

In general, the algorithm consists of three steps: taking regions unknown to the cache; adding invalidated regions (in the unstable case); and removing the dominance regions of cached skyline points. Algorithm 1 presents the pseudocode (with unstable case handling omitted due to space constraints).

Lines 2–10 calculate the *overlap region*, $o = \langle \underline{o}, \overline{o} \rangle$, the area satisfying both C and C', by splitting the space into sections based on the boundary of the cache item for each dimension, eventually yielding the *overlap region* and disjoint regions around it. In the stable case o can simply be removed (Line 11); we discuss the unstable case later. Line 12 discards any hyper-rectangle $h = \langle \underline{h}, \overline{h} \rangle$ for which $\underline{h} = \overline{o}$, since h is clearly in a dominance region. After Line 12, the first of the three steps is complete, and $\mathcal{H}$ captures

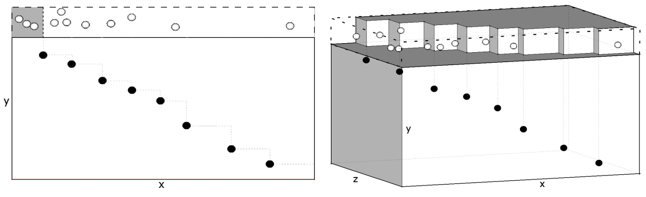
Algorithm 1 MPR - Stable $Sky(S, C)$ relative to C'

```
Input: $I = \langle Sky(S, C), MBR, C \rangle, C'$
Output: A set of range queries
 1: $\mathcal{H} \leftarrow$ Set of hyperrectangles (Initially only $R_{C'}$)
 2: for all dimensions $i \in D$ do
 3:     for all hyperrectangles $r \in \mathcal{H}$ do
 4:         Copy $r$ to $r_{\leq}$, $r_{\cap}$, and $r_{\geq}$
 5:         Add to $r_{\leq}$ constraint $p[i] \leq \underline{C}[i]$
 6:         Add to $r_{\cap}$ constraints $\underline{C}[i] \leq p[i] \leq \overline{C}[i]$
 7:         Add to $r_{\geq}$ constraint $\overline{C}[i] \leq p[i]$
 8:         Del $r$ from $\mathcal{H}$ and, if satisfiable, add $r_{\leq}, r_{\cap}$ and $r_{\geq}$
 9:     end for
10: end for
11: Remove overlap region $o = (R_C \cap R_{C'})$ from $\mathcal{H}$
12: Remove $h \in \mathcal{H}$ where $\underline{h} = \overline{o}$.
13: for all skyline points $u \in Sky(S, C)$ do
14:     for all dimensions $i \in D$ do
15:         for all $r \in H$ not marked with $u$ and $DR(u) \cap r \neq \emptyset$ do
16:             Copy $r$ to $r_{\leq}$ and $r_{\geq}$
17:             Add to $r_{\leq}$ constraint $p[i] \leq u[i]$
18:             Add to $r_{\geq}$ constraint $p[i] \geq u[i]$
19:             Mark $r_{\leq}$ with $u$ and flag $r_{\geq}$ as dominated
20:             if $r_{\leq}$ and $r_{\geq}$ are satisfiable then
21:                 Remove $r$ and add $r_{\leq}$ and $r_{\geq}$ to $\mathcal{H}$
22:             end if
23:         end for
24:     end for
25:     Discard all $r \in H$ flagged as dominated
26: end for
27: Return $H$ as range queries
```

all regions outside the cache in which missing skyline points could exist. Lines 13–26 conduct the third step, looping through each dimension of each skyline point to split the remaining $h \in \mathcal{H}$. With each split we flag one part of h as *processed* by the current point and the other as being *dominated*. The dominated part can be discarded on Line 25, and the flagging of the other part avoids unnecessarily resplitting it. For simplicity of presentation, we assumed no points lie on a range query border. This assumption can be removed by setting either inequality to be strict on Lines 5–7, 16–21. Finally, the unstable case is solved similarly; we simply run a slight modification of Algorithm 1 as a preprocessing step to determine the invalidated regions, and add those to the set $\mathcal{H}$ between Lines 11 and 12. The modification is an inversion of the logic: we want to process (not discard) the overlap region, o, and discard (not keep) the rest. We want to keep (not discard) that which overlaps dominance regions and discard (not keep) the rest. This inverted logic produces a quite small set $\mathcal{H}'$ that exactly represents the unstable, invalidated region. By adding $\mathcal{H}'$ to $\mathcal{H}$ after Line 11, it is then reduced exactly the same as the stable part of the MPR.

5.3 Approximating the MPR

In $2D$ cases (such as Figures 3 and 4a), the MPR (gray region) of the changed constraints (the dashed lines) relative to the old skyline (solid black points) is a simple rectangle. However, each new dimension adds complexity. By considering a third dimension (Figure 4b), the same 8 points and set of constraints produces an MPR consisting of 8 rectangular regions (the hollow part on top). This complexity grows for each distinct z-coordinate because the dominance region of each skyline point is (logically) subtracted from the MPR.



(a) 2D projection (b) 3D projection

Figure 4: More dimensions = more complex dominance regions

Therefore, we introduce the *Approximate MPR* (aMPR). The aMPR is a conservative approximation of the MPR which produces no false negatives by simplifying the structure of MPR, thus creating a structure that decomposes into fewer, but larger, disjoint range queries. This in turn produces a superset of the points in MPR, thus guaranteeing completeness at a lower processing cost. The aMPR represents a middle ground approach between the minimum reads of the MPR and the maximum points read of the naive approach in [3].

The aMPR arises from a simple observation. As mentioned earlier, the complexity of the MPR comes from pruning with many multidimensional dominance regions at once. However, of all skylines points, those nearest to $\underline{C}'$ are likely to prune the most points. (This is the same intuition as for sort-based skyline algorithms [8].) So, we use only the dominance region of a small set of k nearest neighbors (NN) to $\underline{C}'$, rather than all skyline points. Algorithmically, the loop on Line 13 is replaced with the assignment, $u \leftarrow \{\mathrm{NN_1}, \ldots, \mathrm{NN_k}\}$. The optimal number of nearest neighbors to use and the trade-off presented by the approximation is evaluated experimentally in Section 7.

6. CACHE-BASED CONSTRAINED SKYLINE

With the components introduced in Sections 4-5, our Cache-Based Constrained Skyline (CBCS) method works as follows. We assume an in-memory cache with n cache items $\{I_1, \ldots, I_n\}$, organized by an R^*-tree indexing the MBR of each cached skyline. Upon receiving a query $Sky(S, C')$, we perform a search on the R^*-tree fetching all cache items where $R_{C'} \cap \mathrm{MBR} \neq \emptyset$. If none exist, $Sky(S, C')$ is computed naively. If more than one cache item is returned, we select the most efficient based on a cache search strategy (Section 6.1). We then compute the MPR as per Section 5. Finally we fetch the points in the MPR, merge them with the cached $Sky(S, C)$, and compute $Sky(S, C')$.

6.1 Cache search strategies

A cache search strategy takes m query-overlapping cache items $\{I_1, \ldots, I_m\}$ as input and aims to return the cache item most efficient for computation of the query. We suggest several cache search strategies, which we will compare experimentally in Section 7. *Random* chooses a cache item uniformly at random among the m overlapping ones. *MaxOverlap* chooses the cache item with the highest degree of overlap with the query region. *MaxOverlapSP* functions as MaxOverlap, except it prefers cache items whose skyline $Sky(S, C)$ is stable relative to C' even if there is an unstable option with a higher degree of overlap. *Prioritized1D* gives preference to simple cases of single changes (as in Sect. 4.2) as follows: Case 2, Case 3, Case 1, General case stable (i.e. not 1D), Case 4 and General case unstable. Ties are broken using MaxOverlap. The case prioritizes were chosen by experimental evaluation. *PrioritizednD(C1,C2,C3,C4)* generalizes this case-prioritization idea, by independently scoring the four cases (i.e., $C1 \ldots C4$) and penalizing cache items for each dimension where constraints differ from the queried. Initial experiments have shown *PrioritizednD(10,0,5,20)* performs well, thus we use it as *PrioritizednD (Std)*. To demonstrate the importance of proper priorities, we also include a variant *PrioritizednD(10,50,30,0)* denoted *PrioritizednD (Bad)*. Finally *OptimumDistance* chooses the cache item whose lower constraint corner is closest to the lower corner of the queried constraints, to give priority to likely dominating regions.

6.2 Cache replacement & dynamic data

Common cache replacement strategies (i.e., LRU, LCU) are supported by insertion and use counters on the R^* tree. Dynamic data can be supported by viewing each cache item as a separate dataset with a continuous skyline query maintained by any existing method (e.g. [13, 19, 21]). Due to space constraints, evaluation of cache replacement strategies and dynamic data are omitted from Section 7 and left for future work.

6.3 Multiple cache items

As an extension to the work presented in this paper, one might consider exploiting more than one overlapping cache item during processing. Such a strategy could be beneficial given the increased pruning ability from two cache items. However due to the added complexity, more range queries would be generated, cache search strategies would become more complicated and the number of different overlap cases would require elaborate specialized processing methods. These added complexities merit a separate research effort into such a method and thus we leave this for future work.

7. EXPERIMENTAL EVALUATION

In this section, we provide an extensive experimental evaluation of our CBCS method, investigating scalability, the effectiveness of the approximate MPR, and the cache search strategies. We experimentally compare CBCS to the existing *BBS* method for computing constrained skylines, as well as a *Baseline* method that fetches all points in S_C with a range query and applies a standard skyline algorithm (as suggested in [3]). We use the Sort-Filter Skyline (SFS) [8] algorithm in both the *Baseline* method and our own *CBCS* method. While more complex skyline algorithms, e.g., BSkyTree [16], might produce faster overall runtimes, our contribution is orthogonal in that the benefit of our *CBCS* method is independent of the skyline algorithm used, as we show in Section 7.3.

Experiments are performed on Linux with kernel 3.2.0-61-generic, an Intel Core 2 Quad Q8400 2.66 Ghz CPU and 8 GB memory. All algorithms are implemented in Java, using a publicly available Java-based R-tree implementation [2]. Data is stored in PostgreSQL 9.1.13 with each dimension indexed by a standard B-tree. The cache is implemented as a simple in-memory cache, organized through an R^*-tree that indexes the MBR for each cache item.

All methods are evaluated in separation with the DBMS restarted between runs for fair comparison. In preliminary experiments, we also tested a baseline using sequential scan, but it was consistently slower than the baseline using the indexes; so, we omit it for space.

We evaluate with synthetic data by generating independent, correlated and anti-correlated data using the standard generator from [3]. For real data we use a Danish real estate dataset covering almost 4.2 million properties in Denmark as of 2005. The full 2005 dataset is not publicly available but the current 2013 version can be browsed online [3].

[2]http://libspatialindex.github.com
[3]https://www.ois.dk/

7.1 Query workload generation

Existing constrained skyline work does not study sets of queries, but only single queries. We therefore construct a query generator mimicking interactive search patterns as studied also in relation to constrained skyline queries earlier [6, 17]. The generator chooses an initial set of constraints for each $i \in D$ with $\underline{C}[i]$ and $\overline{C}[i]$ set randomly between 0 and 3 standard deviations from the mean of dimension i, modeling that, for example, average-sized houses are most likely to be searched. Subsequent constraint changes are modeled as follows: 1) The dimension to vary is chosen randomly; 2) whether to increase/decrease lower/upper constraints is chosen at random; and 3) a new query is generated from the old, with a $5\% - 10\%$ change in the chosen dimension and direction. Step 3) is repeated $1 - 10$ times to mimic an interactive scenario with one user posing several similar queries. All steps are repeated until the desired number of queries has been generated.

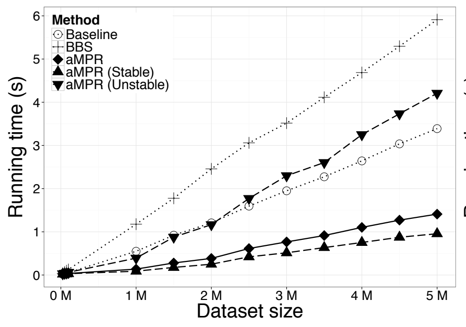
We evaluate all methods with two different query workloads: (1) the aforementioned *Interactive exploratory search* and (2) *Independent* queries in a multi-user system. Workload (1) assumes an empty cache and uses the generator to create 5 independent sets of 100 queries mimicking 5×100 actions in an interactive exploratory search. Workload (2) assumes a preloaded cache with 2000 queries, where we receive a number of new independent single queries each generated like the initial query in the generator.

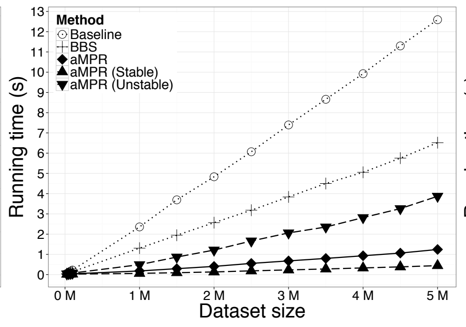
Unless stated otherwise locally, the cache search strategies used are *MaxOverlapSP* for interactive exploratory search queries and *PrioritizednD (Std)* for independent queries.

7.2 Interactive Search - Dataset size & Dimensionality

Figures 5a-5c show the average running time of our *CBCS* using *aMPR* compared to *BBS* and *Baseline*, for increasing dataset size on $5D$ data. Initial experiments showed using 1 NN for *aMPR* gave the most consistently good results for interactive search scenarios. *CBCS* average performance is labeled *aMPR*, further broken down into *aMPR (Stable)* and *aMPR (Unstable)* for performance on stable/unstable cases respectively. Results are averaged over the same 5×100 interactive exploratory search queries. For distributions we see that all methods scale approximately linearly. This is expected since the same range query will require a linear amount of extra processing for each increase in dataset size, regardless of the resulting constrained skyline. By comparing *Baseline* to the *CBCS* methods, we also see that we scale significantly better than the *Baseline* on average for all distributions, especially when the cached skyline is stable relative to the queried constraints in *aMPR (Stable)*. In these cases, a partial skyline result requires fetching only a small subset of what *Baseline* fetches. Thus we both read fewer points and conduct fewer dominance tests. However, while *aMPR* scales well on average and the stable cases in *aMPR (Stable)* scale very well, the unstable cases in *aMPR (Unstable)* fare less well. As discussed in Section 4.2, instability can cause recomputation if a cached skyline point falls outside of the new constraints. Still, the only case in which the *aMPR (Unstable)* does not outscale *Baseline* is on independent data with $\geq 2M$ points.

Interestingly, *BBS* performs worse than *Baseline* in several cases and consistently for independent data. This is most likely due to the overhead in R-tree queries when few entire regions are pruned or included in the skyline. As a final note, observe the scales in Figures 5a-5c differ and that, perhaps surprisingly, correlated data is more of a challenge for the methods than independent data. Broadly speaking independent data is evenly distributed and correlated data is grouped in sections of the dataspace. The same queries that returned a given number of datapoints for the independent data


(a) Independent


(b) Correlated

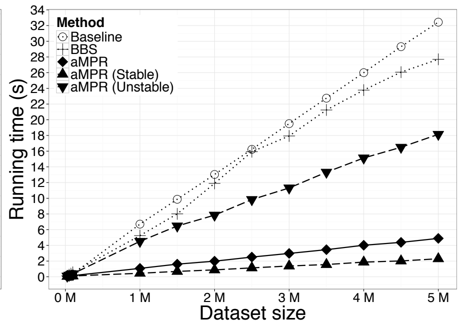

(c) Anti-correlated

Figure 5: Scalability with increasing dataset size for interactive exploratory search queries ($|D| = 5$)

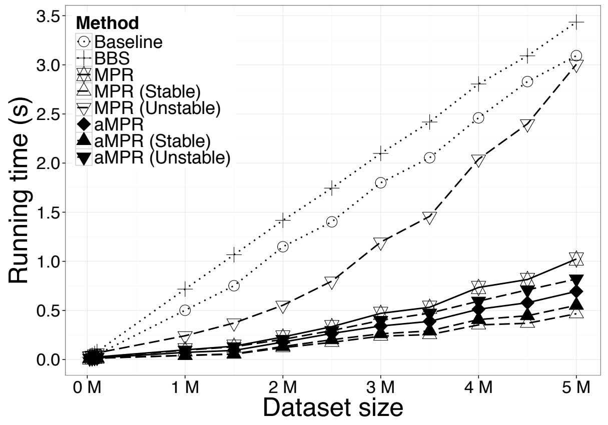


Figure 6: Scalability with increasing dataset size for interactive exploratory search queries (Independent, $|D| = 3$)

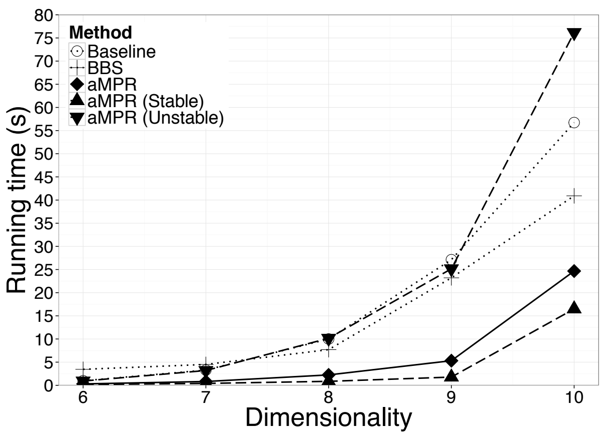


Figure 7: Efficiency with increasing dimensionality ($|S| = 1M$, $|D| > 5$)

can thus return significantly more for correlated data, if the queries happen to cover where the data is most concentrated. This is exactly what we see here, since the average number of points read is 7 times higher for correlated data than for independent data. Note that despite this, the performance of each method is not reduced 7 times, because the computation of the skyline on corrrelated data points is much faster.

Figure 6 shows the same experiment as in Figures 5a-5c but for a 3-, rather than 5-, dimensional independent dataset. We include the exact *MPR* with a stable/unstable split as with *aMPR*. Just as in Figure 5a, we see that *BBS*, *Baseline* and *aMPR* all scale linearly. However while *BBS* performs better, the *Baseline* is still faster for independent data, since the increased efficiency of the R-tree in $|D| = 3$ is equally matched by the benefit of simpler dominance tests in the *Baseline*. The *aMPR* method remains superior to the *Baseline* as in Figure 5a, but due to the decreased dimensionality even unstable cached skylines in *aMPR (Unstable)* are scaling well. Finally the use of the exact *MPR* rather than the *aMPR* means stable cache items yield superior results since *MPR* prunes more of the search space than *aMPR*. However while the same superior pruning applies to unstable cache items, the *MPR* method is significantly slower than the *aMPR*, since cache invalidation yields a prohibitive amount of range queries with subsequent random access latency for *MPR*. We will discuss a further breakdown of these performance factors for *aMPR* and *MPR* in Section 7.3. Finally we observe scalability with regard to $|D|$ in Figure 7. Note that, unlike unconstrained skyline queries, fixing the dimensionality in constrained skylines has some important implications. Depending on the constraints, adding a dimension may in fact increase the efficiency of a constrained skyline query by reducing the input size. In order to avoid such arbitrary effects, we expand the queries from Figures 5a-5c by adding an unconstrained dimension to each query for each dimension over 5. The dimensionality results for $|D| = 8$ are thus constrained on 5 dimensions and unconstrained on 3.

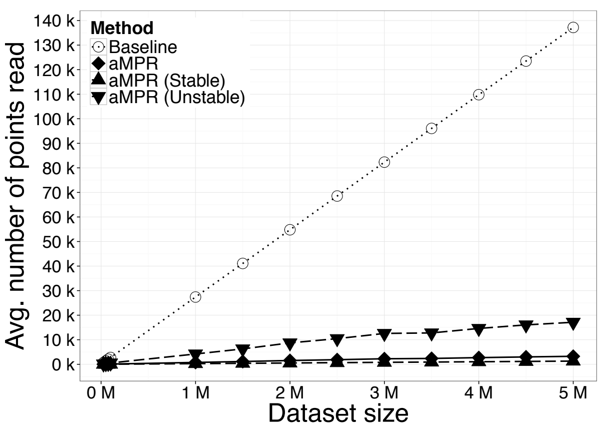
As expected, all methods deteriorate exponentially with $|D|$ as the skyline size increases. For *BBS*, the performance of the underlying R-tree degrades and for the *aMPR*, the number of range queries generated increases (see Section 7.3).

7.3 CBCS performance breakdown

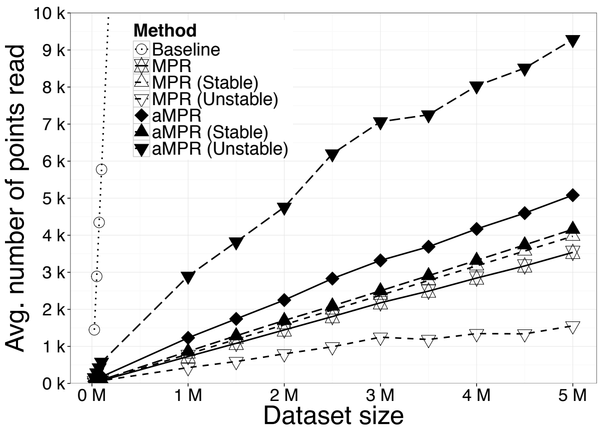
We further analyze *CBCS* by investigating 3 key factors: the number of points fetched, the number of range queries issued, and the types of constraint changes.

7.3.1 Number of points read from disk

Figure 8a shows points read by *Baseline* and *aMPR* for the experiment from Figure 5a. The number of points read by *Baseline* increases significantly with dataset size, while the increase for *aMPR* is limited except for the unstable cases in *aMPR (Unstable)*. This is key to the performance of *aMPR* since the number of points read is primarily influenced by the difference in cardinality between sets

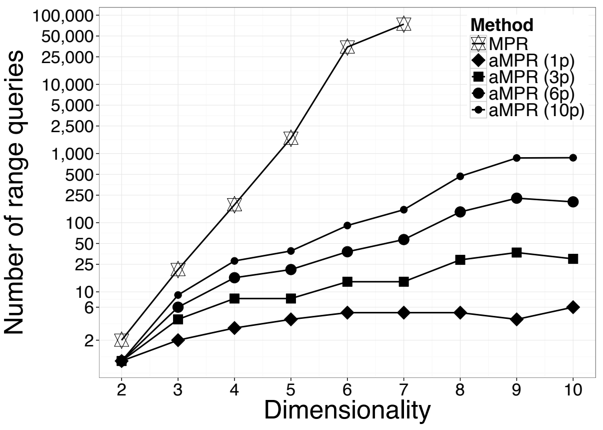


(a) $|D| = 5$

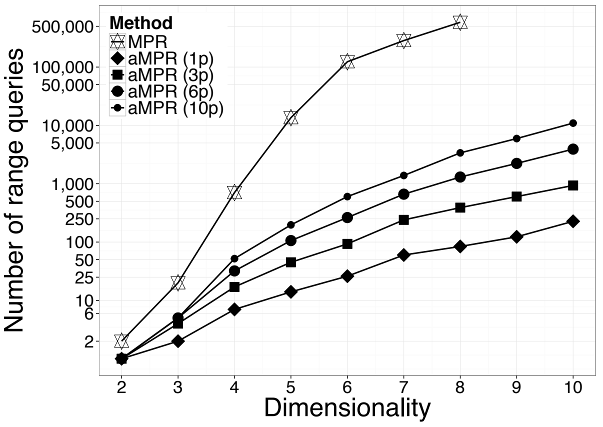


(b) $|D| = 3$

Figure 8: Avg. number of points read (Independent, $|S| = 1M$)



(a) Interactive exploratory search queries



(b) Independent queries

Figure 9: Avg. number of range queries generated (Independent, $|S| = 5k$)

S_C and $S_{C'}$, rather than the actual size of each set. The larger increase for *aMPR (Unstable)* arises both from the likelihood of cached skyline points being outside the queried constraints and increasing the number of new points included due to invalidation inside the cache item.

Figure 8b which shows the average number of points read from disk for *Baseline*, *aMPR* and *MPR* in comparison, corresponding to Figure 6. The same pattern as in Figure 8a can be seen for *Baseline* relative to *aMPR*, while *MPR* consistently reads fewer points than *aMPR* from disk, since it is the minimal set. While an unstable cached skyline yields relatively many reads for *aMPR*, the exact opposite is the case for *MPR*. This illustrates that while *computing* the exact invalidation of a cache item is computationally expensive, the *extent* of invalidation is limited.

7.3.2 Number of range queries generated

Figure 9 shows the average number of range queries for *MPR* and *aMPR* with 1,3,6 and 10 NNs on interactive (Figure 9a) and independent (Figure 9b) workloads. Both graphs use logarithmic scales and the dataset is limited to $5k$ points so that we can scale *MPR* to higher dimensions. Figure 9a reveals that the exact *MPR* rapidly generates extra queries as $|D|$ increases, e.g., a $6D$ query/cache item pair generates almost $50k$ disjoint range queries to cover the MPR. This number would be even higher for $> 5K$ points. For the *aMPR*, the reduced hyperplane splitting, while increasing the number of points read, greatly decreases the amount of queries generated dependent on the amount of NNs used. Note that we did not include results for *MPR* for dimensionalities 8,9 and 10, since just generating the range queries here took several hours for each query. Thus the approximation really improves scalability as $|D|$ increases.

Figure 9b confirms these trends for independent queries as well, but both methods generate more queries and the increase is more rapid, because the queries generated overlap less in higher dimensions. Observe that the number of NNs can be used to manipulate the tradeoff between reading few points from disk and decreasing random access. While large quantities of range queries seem problematic, they do not necessarily deteriorate the performance of the method. Considering e.g. Figure 9b for $|D| = 4$ and $\#NN = 10$, on average ≈ 61 queries were generated for *aMPR*, but the number actually reading data only averaged 33. The remaining queries were discarded by the DBMS without any disk seeks because the B-trees detect the empty queries. For $|D| = 10$ and $\#NN = 10$ these numbers increase to 13353 queries generated of which only 114 read data from disk.

Note that with only $5K$, no stable cases were generated for $|D|$>4. From Theorem 1, this is not surprising, since just one dimension i where $\underline{C}'[i] > \underline{C}[i]$ causes instability of $Sky(S, C)$ relative to C'.

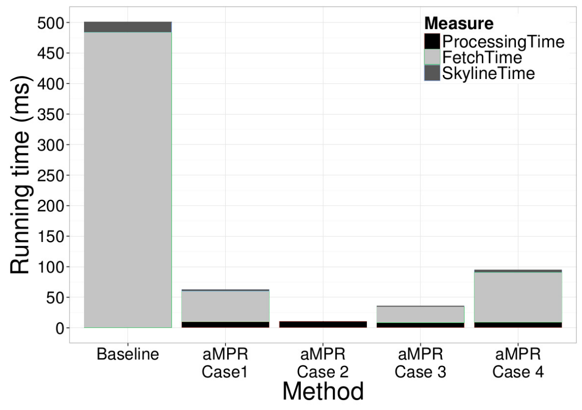


Figure 10: Avg. ms per stage (Independent, $|S| = 1M$, $|D| = 3$)

So, for independent queries, *CBCS* methods work best for lower dimensional settings, and shows most benefit for exploratory queries.

7.3.3 Types of constraint changes

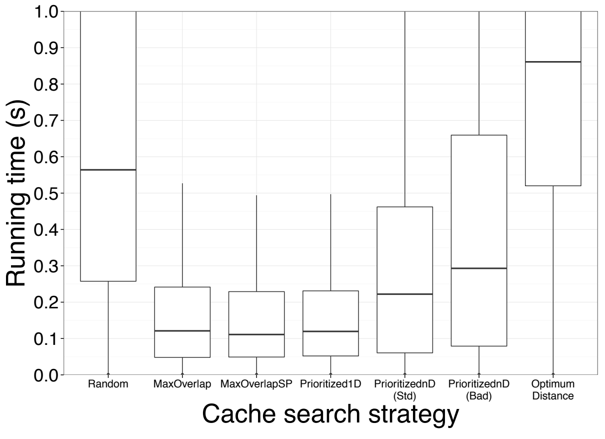
Figure 10 breaks down computation for $1M$ points (settings as in Figure 6) into 3 stages: processing, fetching, and skyline computation, corresponding to the main-memory selection of range queries, the latency to read points from disk, and the running of SFS, respectively. *Baseline* has no processing stage, but suffers long fetching. Conversely, *aMPR* has light processing, which reduces fetching and then, having fewer input points, skyline computation. Considering specific types of changes, *aMPR Case 2* has no fetching stage or computation stage, since this is a simple case which only requires removing cached skyline points. *aMPR Case 3* shows a slight processing stage followed by a significantly smaller fetching stage than both *Baseline* and *aMPR Case 1* since we are able to prune the search space significantly using cached skyline points. Note that while the relative gains of the fetching stage from *aMPR Case 1* to *aMPR Case 3* are only half, a larger portion of points is pruned, with random access being more time consuming.

To conclude, we see that the superior performance of *aMPR* and *MPR* arises primarily from the reduced reads from disk, which reduces both fetching and skyline computation. Also, the performance is independent of the skyline algorithm used, since this is anyway not a bottleneck. Finally we see that the *MPR* requires too many range queries for mid- to high-dimensional data, but the *aMPR* generates a small, stable number of range queries independent of the dataset size.

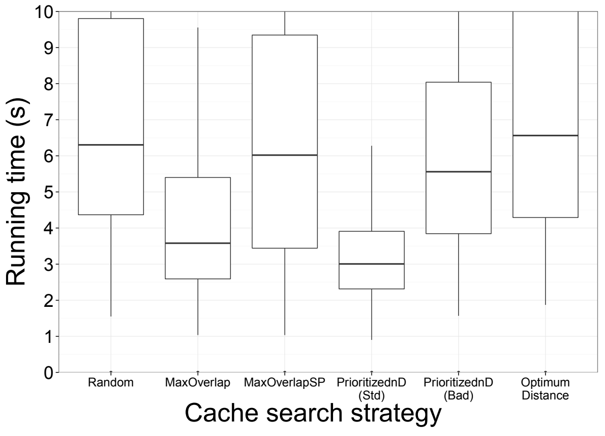
7.4 Cache search strategies

Our last synthetic experiment shows the distribution of response times for each proposed cache search strategy from Section 6.1, using *aMPR* on 5×100 interactive queries (Figure 11a) and 500 independent queries (Figure 11b).

Compared to the *Random* strategy we can see there is a clear benefit in using overlap as a guiding factor (observe *MaxOverlap*, *MaxOverlapSP* and in part *Prioritized1D* which uses *MaxOverlap* to settle ties). High overlap yields smaller MPRs, especially in stable cases. On the other hand, prioritizing only stable cases is not a good strategy for independent queries, as is clear from *MaxOverlapSP*: such queries are likely to vary in several dimensions such that choosing solely on stability may select an item with poor overlap or many changed dimensions. Instead a balanced ranking-based approach like *PrioritizednD (Std)* is most promising, since it not only considers stability but also case complexity. *PrioritizednD*



(a) Interactive exploratory search queries



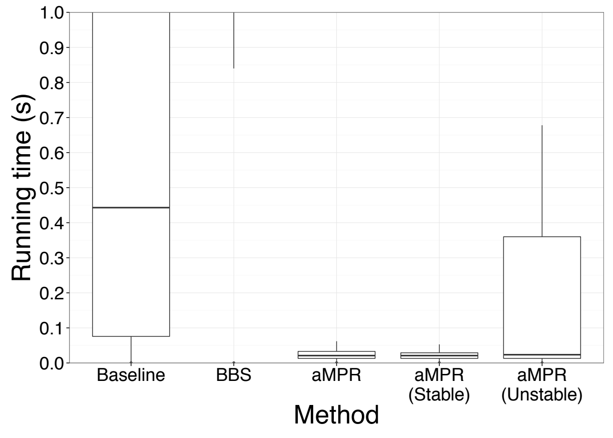
(b) Independent queries

Figure 11: Cache search strategies ($|S| = 1M$, $|D| = 5$)

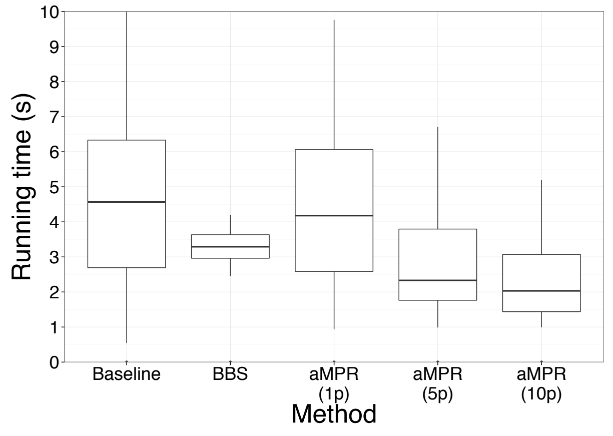
(Bad) shows poor performance demonstrating that the case-based scoring is important for performance. Finally, *OptimumDistance* performs poorly: considering only closeness in terms of dominance fails to capture the complexity of cases and overlap.

7.5 Real data

We study real data covering ~ 4.2 million properties in Denmark as of 2005 using 4-dimensions suitable for constrained skyline computation: *year* (year of construction), *sqrm* (size in m^2), *valuation* (property tax valuation) and *price* (actual sales price). The final dataset size is $1.28M$ records after removing records with missing data. Figures 12a and 12b respectively show the distribution of response times for 10×100 exploratory search queries and 50 independent queries. Figure 12a shows our *aMPR* method is superior to both *Baseline* and *BBS*, with *BBS* managing about 2.2 seconds on average per query and *Baseline* performingly significantly better at about 0.45 seconds. Note the average response time of *aMPR (Unstable)* is actually low due to limited invalidation, while the worst case invalidation yields response times above the average for *Baseline*. Figure 12b shows a set of independently generated queries. Here *Baseline* varies heavily in performance due to varying query selectivities, while *BBS* is stable with the changing constraints. The remaining 3 plots show *aMPR* with varying numbers of NNs. The number of NNs chosen in this case has a large impact on performance, since using only 1 as with the ex-



(a) Interactive exploratory search queries



(b) Independent queries

Figure 12: Performance on Danish property data ($|S| = 1.28M$, $|D| = 4$)

ploratory queries yields poorer results than *BBS*, while 5 NNs or 10 NNs greatly outperforms it. Using more than 10 NNs however did not provide any significant further benefit in this case.

In conclusion, the major performance factors are the number of disk reads performed and the degree of random access due to multiple range queries. We have shown that *CBCS* performs substantially better on related queries than existing methods. The performance benefit remains stable for increasing dataset size and dimensionality, when using approximate *aMPR*. The cache search strategy is a clear determinator of the resulting performance with the main factors being *average overlap*, *stability* and *case distributions*.

8. CONCLUSION

In this paper we introduced a novel method for computing constrained skylines using an in-memory cache. Our method was envisioned under two common types of query workloads with related queries, namely interactive search and multi-user systems. We determined four possibilities for incremental query/cache overlap, analyzed them and presented specialized techniques for each. For general query/cache overlap, we introduced the *Missing Points Region*, which minimizes points read from disk by exploiting the cache item's relation to the query. To increase the practical performance of this general method, we introduced a conservative approximation of the MPR, called aMPR, to balance the trade-off between resultant range queries and points read from disk. Finally we introduced a set of heuristics to choose the most efficient overlapping cache item for a query. Our extensive experimental evaluation revealed, among other things, that the choice of cache item to use for processing has a big impact on performance and that our method significantly outperforms existing approaches when related queries are present.

9. ACKNOWLEDGMENTS

This research was supported in part by the Danish Council for Strategic Research, grant 10-092316.

10. REFERENCES

[1] K. M. Banafaa and R. Li. Efficient algorithms for constrained subspace skyline query in structured peer-to-peer systems. In *Proc. WAIM*, 2012.

[2] A. Bhattacharya, B. P. Teja, and S. Dutta. Caching stars in the sky: a semantic caching approach to accelerate skyline queries. In *Proc. DEXA*, 2011.

[3] S. Börzsönyi, D. Kossmann, and K. Stocker. The Skyline operator. In *Proc. ICDE*, 2001.

[4] S. Chaudhuri, N. Dalvi, and R. Kaushik. Robust cardinality and cost estimation for skyline operator. In *Proc. ICDE*, 2006.

[5] L. Chen, B. Cui, L. Xu, and H. Shen. Distributed cache indexing for efficient subspace skyline computation in p2p networks. In *Proc. DASFAA*. 2010.

[6] S. Chester, M. L. Mortensen, and I. Assent. On the suitability of skylines queries for data exploration. In *Proc. ExploreDB*, 2014.

[7] S. Chester, D. Sidlauskas, I. Assent, and K. Bøgh. Scalable parallelization of skyline computation for multi-core architectures. In *Proc. ICDE*, 2015.

[8] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting: Theory and optimizations. In *Intelligent Information Systems*, 2005.

[9] B. Cui, H. Lu, Q. Xu, L. Chen, Y. Dai, and Y. Zhou. Parallel distributed processing of constrained skyline queries by filtering. In *Proc. ICDE*, 2008.

[10] E. Dellis, A. Vlachou, I. Vladimirskiy, B. Seeger, and Y. Theodoridis. Constrained subspace skyline computation. In *Proc. CIKM*, 2006.

[11] M. Endres and W. Kiessling. Semi-skyline optimization of constrained skyline queries. In *Proc. ADC*, 2011.

[12] K. Fotiadou and E. Pitoura. BITPEER: continuous subspace skyline computation with distributed bitmap indexes. In *Proc. DAMAP*, 2008.

[13] Y.-L. Hsueh, R. Zimmermann, and W.-S. Ku. Efficient updates for continuous skyline computations. In *Proc. DEXA*, 2008.

[14] Y.-L. Hsueh, R. Zimmermann, and W.-S. Ku. Caching support for skyline query processing with partially-ordered domains. In *Proc. SIGSPATIAL*, 2012.

[15] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *Proc. VLDB*, 2002.

[16] J. Lee and S.-w. Hwang. Scalable skyline computation using a balanced pivot selection technique. *Inf. Syst.*, 39, 2014.

[17] M. Magnani, I. Assent, K. Hornbæk, M. R. Jakobsen, and K. F. Larsen. SkyView: A user evaluation of the skyline

operator. In *Proc. CIKM*, 2013.

[18] Y. Mass, M. Ramanath, Y. Sagiv, and G. Weikum. Iq: The case for iterative querying for knowledge. In *Proc. CIDR*, 2011.

[19] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *ACM Trans. Database Syst.*, 30, 2005.

[20] D. Sacharidis, P. Bouros, and T. Sellis. Caching dynamic skyline queries. In *Proc. SSDBM*, 2008.

[21] P. Wu, D. Agrawal, O. Egecioglu, and A. El Abbadi. Deltasky: Optimal maintenance of skyline deletions without exclusive dominance region generation. In *Proc. ICDE*, 2007.

[22] L. Zhang, Y. Jia, and P. Zou. A grid index based method for continuous constrained skyline query over data stream. In *Advances in Web and Network Technologies, and Information Management*. 2009.

[23] S. Zhang, N. Mamoulis, and D. W. Cheung. Scalable skyline computation using object-based space partitioning. In *Proc. SIGMOD*, 2009.

APPENDIX

Proof of Theorem 1. Consider left [L] and right [R] sides of the OR expression. From [R] we have $(\exists i \in D\colon \underline{C}'[i] > \overline{C}[i] \vee \overline{C}'[i] < \underline{C}[i]) \implies R_C \cap R_{C'} = \emptyset$, thus $Sky(S,C)$ is stable in this case. For [L] we prove $(\forall i \in D\colon \underline{C}'[i] \leq \underline{C}[i])$ implies stability, by contradiction. Assume $\exists s \in Sky(S,C')\colon (s \in S_C) \wedge (s \notin Sky(S,C))$. From Def 1, this implies $\exists t \in S_C\colon t \succ s$. This in turn means $s \in Sky(S,C') \implies t \notin S_{C'} \implies \exists i \in D\colon \overline{C}'[i] < t[i] \leq s[i] \leq \overline{C}[i]$, i.e. t and s both do not satisfy constraints C' and thus $s \notin Sky(S,C')$ contradicting the assumption. Observe that [L] and [R] are the only situations with guaranteed stability, since cases not satisfying Thm 1 must have $R_C \cap R_{C'} \neq \emptyset$ and $\exists i \in D\colon \underline{C}[i] < \underline{C}'[i] \leq \overline{C}[i]$. Thus for $u \in Sky(S,C)$, $\underline{C}[i] \leq u[i] < \underline{C}'[i] \leq \overline{C}[i]$ we could have $v \in S_C$, $u \succ v$ and $\underline{C}[i] \leq u[i] < \underline{C}'[i] \leq v[i] \leq \overline{C}[i]$. Since $u \notin S_{C'}$ we might then have $v \in Sky(S,C')$, making $Sky(S,C)$ unstable relative to C'. $\square$

Proof of Theorem 2. Since $Sky(S,C)$ is stable relative to C' given Thm 1, equality follows from Cor 1. Minimality holds since $\nexists s \in Sky(S,C), t \in S_{\Delta C}\colon s \succ t$. $\square$

Proof of Theorem 3. Observe $S_{C'} \subset S_C$ and that $Sky(S,C)$ is stable relative to C' given Thm 1. Thus we must have $Sky(S,C') \subseteq Sky(S,C)$ and $Sky(S,C') = Sky(S,C) \cap S_{C'}$ follows. Minimality holds since the reduction simply removes cached skyline points and no further reads are necessary. $\square$

Proof of Theorem 4. Observe $Sky(S,C)$ is stable relative to C' given Thm 1. Thus equality follows from Cor 1 since for $s \in Sky(S,C')$ we either have $s \in Sky(S,C)$ or $s \in S_{\Delta C}$, where $\nexists t \in Sky(S,C)\colon t \succ s$.

Minimality holds since $\forall u \in Sky(S,C), v \in (Sky(S,C) \cup \{s \in S_{\Delta C} \mid \nexists t \in Sky(S,C)\colon t \succ s\}\colon u \not\succ v)$, i.e. we have no further known points to prune $S_{\Delta C}$ with. $\square$

Proof of Theorem 5. Given Thm 1, $Sky(S,C)$ may be unstable relative to C' and we observe $S_{C'} \subset S_C$. At this point we have two possibilities given Cor 2: [St] $Sky(S,C)$ is stable relative to C', or [Ust] $Sky(S,C)$ is unstable relative to C'. If case [St] holds, then $Sky(S,C') = Sky(S,C) \cap S_{C'}$ since there is no invalidation. If case [Ust] holds, then from Cor 2 we have $\exists t \in Sky(S,C)\colon t \in S_{\Delta C} \wedge \exists s \in (S_C \cap S_{C'})\colon t \succ s \wedge \nexists u \in (Sky(S,C) \cap S_{C'})\colon u \succ s$, i.e. there exists a removed skyline point which has invalidated part of the cache. Since [St] and [Ust] correspond to the two unioned skyline inputs in Thm 5, completeness holds by virtue of Def 1.

To observe that minimality holds, we first note that from Def 1 we must have $\nexists s \in Sky(S,C), t \in (Sky(S,C) \cap S_{\Delta C})\colon s \succ t \implies \forall s \in (Sky(S,C) \cap S_{C'})\colon s \in Sky(S,C')$, i.e. a removed skyline point cannot dominate a remaining skyline point. Thus minimality holds since only $\{s \in (S_C \cap S_{C'}) \mid \exists t \in (Sky(S,C) \cap S_{\Delta C})\colon t \succ s\}$ is affected by instability given Cor 2 and only $Sky(S,C) \cap S_{C'}$ is available for pruning. $\square$

Proof of Theorem 6. We prove equality in right [R] and left [L] directions. For [R] we assume $w \in Sky(S,C')$. We then have two cases: [1] $w \in (R_C \cap R_{C'})$ and [2] $w \in (R_{C'} \setminus (R_C \cap R_{C'}))$, i.e. w is either in the overlapping region between cache and query ([1]) or outside the cache ([2]).

If we have case [1], then given Cor 2 we have $w \in (Sky(S,C) \cap R_{C'}) \vee (\exists t \in (Sky(S,C) \cap (R_C \setminus R_{C'}))\colon w \in DR(t,C) \wedge \nexists u \in (Sky(S,C) \cap R_{C'})\colon w \in DR(u,C'))$, i.e. w is a cached skyline point or a point included due to invalidation. If instead case [2] holds, then we simply have $\nexists u \in S_{C'}\colon u \succ w$ due to Def 1.

Combined we thus have $(w \in (Sky(S,C) \cap R_{C'})) \vee (w \in \{p \in R_{C'} \mid (p \in (R_{C'} \setminus (R_C \cap R_{C'})) \vee \exists t \in (Sky(S,C) \cap (R_C \setminus R_{C'}))\colon p \in DR(t,C)) \wedge \nexists u \in (Sky(S,C) \cap R_{C'})\colon p \in DR(u,C')\})$ which is equivalent to $w \in Sky((Sky(S,C) \cap S_{C'}) \cup (MPR \cap S_{C'}), C')$ by virtue of Def 1.

For the opposite direction, [L], we assume $w \in Sky((Sky(S,C) \cap S_{C'}) \cup (MPR \cap S_{C'}), C')$. We again have two cases: [St] $Sky(S,C)$ is stable relative to C', and [Ust] $Sky(S,C)$ is unstable relative to C'.

If we have case [St], then given Cor 1 we have $(w \in (Sky(S,C) \cap S_{C'})) \veebar (w \in (R_{C'} \setminus R_C))$, i.e. w is either a cached skyline point or outside the cache.

If instead we have case [Ust], then given Cor 1 and 2, we have $((w \in (Sky(S,C) \cap S_{C'})) \veebar (w \in (R_{C'} \setminus R_C))) \vee (w \in (R_C \cap R_{C'}) \wedge (\exists t \in (Sky(S,C) \cap (R_C \setminus R_{C'}))\colon w \in DR(t,C)) \wedge (\nexists u \in (Sky(S,C) \cap R_{C'}))\colon w \in DR(u,C'))$, i.e. w is either a cached skyline point, a point outside the cache or a point included due to invalidation.

Now observe that the region $R_{C'}$ can be expressed as $R_{C'} = (R_C \cap R_{C'}) \cup (R_{C'} \setminus (R_C \cap R_{C'}))$. We now prove by contradiction that any w satisfying the right hand side of Thm 6 must also be in $Sky(S,C')$. So we assume $w \notin Sky(S,C') \implies \exists v \in Sky(S,C')\colon w \in DR(v,C')$ given Def 1. We then have two cases [I] $v \in (R_C \cap R_{C'})$ and [O] $v \in (R_{C'} \setminus (R_C \cap R_{C'}))$.

If [I] then $v \in Sky(S,C) \vee \exists t \in (Sky(S,C) \cap (R_C \setminus R_{C'}))\colon v \in DR(t,C))$ which implies $w \notin Sky((Sky(S,C) \cap S_{C'}) \cup (MPR \cap S_{C'}), C')$ given Def 1 yielding a contradiction.

If [O] then $v \notin (R_C \cap R_{C'}) \implies v \notin (Sky(S,C) \cap R_{C'}) \implies \nexists u \in (Sky(S,C) \cap R_{C'})\colon u \succ v \implies v \in MPR$ given Def 5, also yielding a contradiction. Hence $w \in Sky(S,C')$. $\square$

Proof of Theorem 7. Proof by contradiction. Assume $p \in MPR$ and $p \notin Sky(S,C')$ can be guaranteed. From Def 1, $p \notin Sky(S,C') \implies \exists t \in Sky(S,C')\ :\ t \succ p$. Observe that t must be known before fetching for us to prune with it, thus $t \succ p \implies t \in (Sky(S,C) \cap R_{C'}) \implies t \in Sky(S,C)$, i.e. t is part of the cached skyline. By the definition of MPR (Def 5), this contradicts our assumption since all $p \in R_{C'}$ where $\exists u \in (Sky(S,C) \cap R_{C'})\colon p \in DR(u,C')$ are excluded from MPR. Thus the MPR is minimal. $\square$

Explanations for Skyline Query Results

Sean Chester and Ira Assent
Data-Intensive Systems Group
Aarhus Universitet, Åbogade 34, 8200 Århus N, Denmark
schester@cs.au.dk   ira@cs.au.dk

ABSTRACT

Skyline queries are a well-studied problem for multidimensional data, wherein points are returned to the user iff no other point is preferable across all attributes. This leaves only the points most likely to appeal to an arbitrary user. However, some dominated points may still be interesting, and the skyline offers little support for helping the user understand why some interesting points are omitted from the results. In this paper, we introduce the *Sky-not query*. Given a query point p, a dataset $\mathcal{S}$, and constraints with bounding corners q_{L} and q_{U}, the Sky-not query returns the alternative constraints q'_{L} closest to q_{L} for which p is in the skyline. This equips the user with an understanding of not just that a point was dominated, but also how severely. He can then assess himself whether the point is competitive.

We first propose theoretical results that show how to drastically reduce the input processed by a Sky-not query, independent of any algorithm. We then offer a skyline-like and an efficient recursive algorithm for solving Sky-not queries, which we evaluate in an extensive experimental evaluation.

1. INTRODUCTION

When exploring unfamiliar data, the *skyline operator* [3] can identify balances among multiple (possibly conflicting) attributes. It selects only those data points for which no other point is preferable across all attributes. Consider the canonical example of selecting a hotel, given the fictitious ones in Table 1. Budget Sleepz (B) is both cheaper and closer to the beach than Cozy Cabin (C); so, it is said to *dominate* the latter. The skyline is exactly and only those points not dominated by any others, in this case not including Cozy Cabin, but including the remainder: $\{\mathrm{A}, \mathrm{B}, \mathrm{D}, \mathrm{E}\}$. For a user, however, this indicates only that C is dominated by some other point, not which, nor how severely. C may be a reasonably competitive choice, even if it is dominated.

More generally, a (skyline) query typically includes range constraints (i.e., a `WHERE` clause) for more sophisticated expression. The user searching for hotels may want the most affordable option, but not something "cheap." Or, he may want a location remote from the city, but reachable within a few hours. Setting these constraints perfectly can be a challenging and iterative process, since notions such as "cheap" and "a few hours" are intentionally under-specified by the user and data-dependent. Yet, the constraints can have substantial impact on the query results, both adding and removing skyline points [6]. For the user, there is no support for understanding why data points are dominated, nor by how much. Especially when comparing several similar queries, the differences can be quite perplexing [18].

ID	Name	Price/nt	Distance
A	Abode Abroad	45	2100m
B	Budget Sleepz	30	4200m
C	Cozy Cabin	40	4500m
D	Diamond Harbour Inn	175	300m
E	Extravagansium	325	100m

Table 1: A fictitious city's hotel offerings.

Furthermore, not all decision criteria are necessarily reflected in the data attributes. Other factors might contribute to preferring a non-skyline point. For example, a user may favour Cozy Cabin on recommendation from a friend or because of a successful advertising campaign. In such a case, the skyline may do a disservice by hiding results that the user expects, ones that match the constraints of his search and are valid options from his perspective. In this paper, we introduce *explanations* for data points excluded by skyline queries. The user can then appraise whether the extra 10 euro per night and 7% from the beach is a tolerable trade-off and adjust his query accordingly, if he wants.

Figure 1 illustrates the technical problem. Given range constraints $(q_{\mathrm{L}}, q_{\mathrm{U}})$, one retrieves a skyline of just hotels A and B, yet C also satisfies the constraints. How does one explain to the user C's absence? In other words, why is C dominated? Similarly to other questions of *why-not provenance* [4, 11, 19, 24], we wish to report an alternative query (i.e., new constraints) so that C is no longer omitted from the results. We find a new position (one candidate is denoted by a star in the figure) for the lower constraints, q_{L}, so that all points dominating C are eliminated. This illuminates the relationship of point C to the rest of the data and, furthermore, concretely recommends to the user a potentially better-suited query.

However, finding the best explanation—the best new position for q_{L}—is non-trivial: anywhere within the rectangle

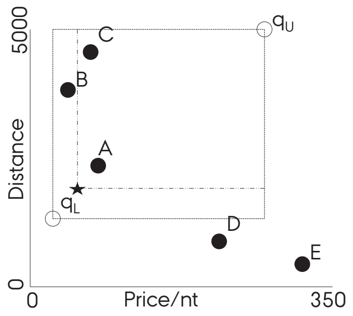


Figure 1: Hotels from Table 1 mapped to points in the plane, shown with a constrained region (q_L, q_U), enclosed by dashed lines. A Sky-not query for point C relocates q_L to a position, such as the star, after which C is in the skyline.

(q_L, C) could potentially improve C's query fate, since the only things that are obvious (later in this paper) is that q_L's old position should dominate its new position (so that every change is an improvement) and that the new position must still dominate C (so that C satisfies the constraints). The best position is the one closest to the original position but that promotes C to the skyline.

In this paper, we introduce and propose solutions for this novel query, the *Sky-not Query*, which quantifies *why* a query point p is *not* in a *sky*line. Algorithmically, we introduce a recursive algorithm that prioritizes cases that favour our pruning rules to efficiently relocate q_L. In comparison to a baseline algorithm and an adaptation of ideas from [12, 14, 17, 21], we show in a detailed experimental evaluation that the Sky-not query can be computed very efficiently.

Contributions and Outline

In this paper, we make the following contributions:

- we introduce the Sky-not Query to improve the useability of the skyline operator (Section 2);
- we derive theoretical properties of Sky-Not Queries, including duality with the SkyDist problem studied in [12,14], that lead to algorithm-independent improvements in efficiency (Section 4); and
- we give two novel algorithms, BRA and PrioReA, based on theoretical analysis (Section 5) for achieving impressive empirical performance (Section 6).

Additionally, we survey related literature in Section 3 and conclude in Section 7.

2. THE SKY-NOT QUERY

In this section, we introduce the novel *Sky-not query* for explaining the absence of a given point in a skyline result. We first recall some general concepts from literature about skylines (Definitions 1-4), before presenting the problem definition (Definition 5).

Generally, we assume a dataset of records represented by a set of Euclidean points, $\mathcal{S}$. For example, the hotels from Table 1 are represented as points in Figure 1. Given $\mathcal{S}$, we denote the dimensionality of the Euclidean space by d and the i'th attribute of a point $s \in \mathcal{S}$ by $s[i]$. Without loss of generality and for the sake of simplifying prose, we assume all attribute values are in the range $[0, M)$, for some positive real, M, and that smaller values are preferable.[1]

2.1 Constrained Skylines

To define the skyline formally, we first define *dominance* (Definition 1). One point *dominates* another if it is at least as desirable in every dimension, and strictly more desirable in at least one. Formally:

Definition 1 (Dominance [3])**.**
Point s *dominates* point t, denoted $s \prec t$, iff

$$\forall i \in [0, d), s[i] \leq t[i] \text{ and } \exists j \in [0, d) \text{ such that (s.t.) } s[i] < t[i].$$

If neither s dominates t nor t dominates s, then s and t are *incomparable*, denoted $s \prec\succ t$. The relation $s \preceq t$ denotes that either $s \prec t$ or $\forall i \in [0, d), s[i] = t[i]$.

For example, from Table 1, Hotel B dominates Hotel C because it is both cheaper and nearer (B is to the lower-left of C in Figure 1). A stronger case is strict dominance (Definition 2), in which strict equalities are used:

Definition 2 (Strict dominance)**.**
Point p *strictly dominates* point q, denoted $p \prec\!\!\prec q$, iff $\forall i \in [0, d), p[i] < q[i]$.

Given range constraints, where q_L denotes the lower bound (0 if unspecified) on every attribute and q_U denotes the upper bound on every attribute (M if unspecified), we call the subset of points satisfying the constraints the *constrained dataset* (Definition 3):

Definition 3 (Constrained Dataset)**.**
Given a set of points, $\mathcal{S}$ and a constraint region (q_L, q_U), the *constrained dataset*, denoted $\mathcal{S}_{(q_L, q_U)}$, is the set of points $\{s \in \mathcal{S} : q_L \prec\!\!\prec s \prec\!\!\prec q_U\}$. We say that each point $s \in \mathcal{S}_{(q_L, q_U)}$ is *inside* the constraint region.

For example, given the constraints in Figure 1, only A, B, C are in the constraint region. The skyline (Definition 4) is the set of points inside the constraint region that are not dominated by any other points inside the constraint region:

Definition 4 ((Constrained) Skyline Query($\mathcal{S}$) [3])**.**
Given a set of points, $\mathcal{S}$, and a constraint region, (q_L, q_U), a *skyline query* returns the set:

$$\text{SKY}(\mathcal{S}, (q_L, q_U)) = \{s \in \mathcal{S}_{(q_L, q_U)} : \nexists t \in \mathcal{S}_{(q_L, q_U)}, t \prec s\}.$$

The set $\text{SKY}(\mathcal{S}, (q_L, q_U))$ is called the *skyline* of $\mathcal{S}_{(q_L, q_U)}$.

For example, in Table 1, with $q_L = (0, 0)$ and $q_U = (350, 5000)$, Hotel C is dominated by Hotel B, but all other hotels are incomparable to each other. Therefore, the skyline is $\{A, B, D, E\}$. On the other hand, with constraints $q_L = (35, 1000)$ and $q_U = (250, 5000)$, as shown with the outer, dotted rectangle in Figure 1, only A and C satisfy the constraints and neither dominate each other; so, they are both in the skyline: $\{A, C\}$.

2.2 Problem Definition

We now define the *Sky-not query* (Definition 5). Informally, given a constrained skyline instance and a query point p, a Sky-not query, denoted $\text{SN}(\mathcal{S}, p, (q_L, q_U), \Delta)$ determines the minimum cost change to q_L after which p would be in the skyline. Formally:

[1]The first assumption is justified by normalization and the second assumption can hold by multiplying values by -1 where maximization is instead preferred.

Definition 5 (Sky-not Query, $\text{SN}(\mathcal{S}, p, (q_\text{L}, q_\text{U}), \Delta)$)**.** Given $\mathcal{S}, p \in \mathcal{S}, (q_\text{L}, q_\text{U})$, s.t. $q_\text{L} \prec\!\!\!\prec p \prec\!\!\!\prec q_\text{U}$, distance function, $\Delta : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^+$, [2]a *Sky-not Query* returns point:

$$q'_\text{L} = \text{argmin}_{q \in (q_\text{L}, q_\text{U}), p \in \text{SKY}(\mathcal{S}, (q'_\text{L}, q_\text{U})} \Delta(q_\text{L}, q).$$

Again considering Figure 1, we could move q_L to the starred position, and then C is in the skyline (i.e., the starred position *solves* the Sky-not query). However, this is not the optimal solution: $q'_\text{L} = (30, 1000)$ also solves the Sky-not query, but is closer to q_L than the starred position.

By learning the Sky-not query result, q'_L, a user can immediately understand how competitive p is relative to the skyline and evolve his subsequent queries accordingly.

3. RELATED WORK

The Sky-not Query (Section 2) ties together a couple of active research topics, which we survey in this section.

Skyline The skyline operator was introduced [3] with two disk-based algorithms, a block-nested loop (BNL) and a partitioning-based (D&C) approach. Since, BNL has been improved with the use of pre-sorting [8] and early termination conditions [1]. The D&C algorithm has been improved with progressively better partitioning schemes [15, 27]. Also, new index-based algorithms based on B-Trees [23] and R-Trees [20] have been proposed. Of these algorithms, BSkyTree [15] reports the best performance; although, this can be improved using multiple processing cores [7].

Although the skyline was proposed in the context of a general SQL query with a `WHERE` clause [3], Papadias et al. [20] were the first to propose algorithms specifically for handling constraints, a problem they term the *constrained skyline* and the subject of study here. The presence of constraints makes the skyline operator much more flexible and practical, because most real queries involve a `WHERE` clause.

Although the skyline is considered as a tool for interactivity, say in discovering user preferences, little research has considered this aspect. Literature on "interactive skylines" (e.g., [16]) seeks to learn user preferences. Our perspective on interactivity here is in helping the user understand *the data*, not their preferences over the attribute domains. To this end, existing work is quite limited. Magnani et al. [18] conducted a user study that showed users are perplexed when posing consecutive queries if new dominance relationships cause previous query results to disappear. Chester et al. [6] conducted an experimental study of how much the skyline changes when users evolve their constraints. Both these works assess the impact on the user of interacting with the skyline, but neither provide solutions for helping the user to evolve queries towards his/her objective.

Why-not Queries Providing the user with an explanation for a missing answer [4,11,19] is a relatively new goal in database research. The general idea of *why-not queries* is to provide a user, who has a specific solution record in mind, with specific details into the cause for its being omitted from the results. Why-not queries have been studied for relational (i.e., SPJUA, sort-project-join-union-aggregate) queries [2, 10], top-k queries [9], and reverse skyline queries [13],[3] but the definitions and techniques do not straight-forwardly apply for skyline queries because each query type excludes candidate results for quite different reasons (no common join key, low weighted sum, & dominance).

However, we do adapt two key ideas from other why-not query types. Because explanations can be arbitrarily complex, it is preferable to determine minimal explanations [28]. Our experiments (Section 6) reveal that Sky-not queries favour explanations involving fewer dimensions changing. Additionally, there are generally two ways of manipulating a query result [24]: either by changing the data or the query parametres. Here, we focus on changing the query, since this is more often under the user's control. With respect to changing the data, some research has gone into manipulating skyline query results [12,14,17,21] from a Business Intelligence perspective, where the objective is to modify one's product offerings in order to penetrate the skyline. We elaborate on the relationship between changing the query and the data in Theorem 5. Lastly, our problem differs from creating competitive products [25]: we aim to determine why *an existing* product is not in the skyline, not to create new (meta-)products from a *collection* of existing products.

Technically related papers We overlap some technical material from tangentially related papers. Our algorithms first find all the points that cause the absence of p in the skyline, then relocate q_L. To detect the causative points, we borrow the idea of *close dominance* [22] (Definition 6). Regarding relocation, Cheema et al. [5] introduce the idea of "safe zones" for dynamic skylines, wherein the safe zone of a point is those query positions in which the point is still a skyline point. Our objective is to find the nearest query point where all safe zones are violated. DeltaSky [26] maintains a view of skyline points to handle deletions.

4. PROPERTIES OF THE SKY-NOT QUERY

In this section, we introduce some algorithm-independent theoretical insights into the Sky-not query. In particular, we show both that the solution space (q_L, p) can be discretized and that the input size can be reduced (Section 4.1). This gives a smaller, finite search space for the problem. We also show duality of the problems of changing the data vs. the query to manipulate a skyline result (Section 4.2). Consequently, techniques for both problems are mutually exchangeable (and, indeed, we do exactly this in our experimental evaluation, Section 6).

4.1 Reduction and Discretization

We start with a couple properties of a Sky-not query solution that are quickly evident from the definitions.

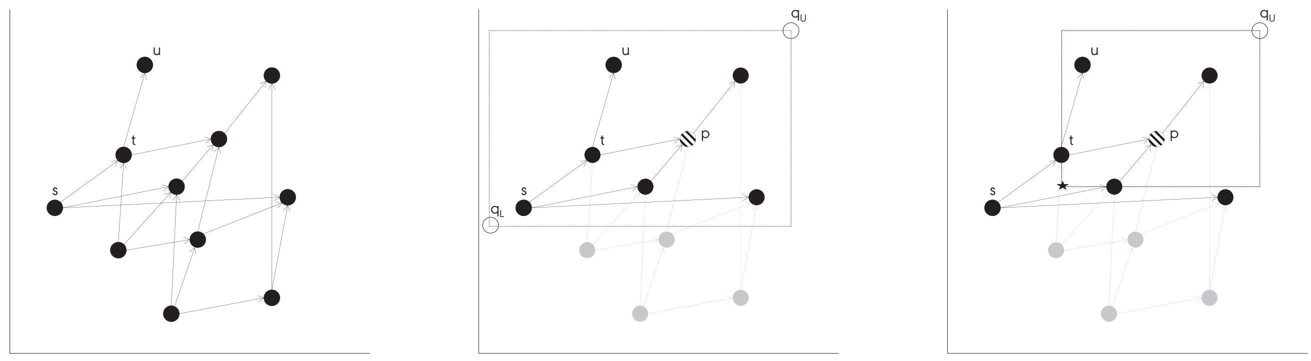
Proposition 1 (Transitive Order)**.** *$q_\text{L} \prec\!\!\!\prec q'_\text{L} \prec\!\!\!\prec p \prec\!\!\!\prec q_\text{U}$.*

Proposition 2 (Undominated Data)**.** *$\forall s \in \mathcal{S}_{(q_\text{L}, q_\text{U})}, s \prec p \implies q'_\text{L} \not\prec s$.*

Proposition 1 simply states that the solution q'_L must necessarily increase or keep the values of q_L and must strictly dominate neither p nor q_U; otherwise, p will not be in the skyline. Proposition 2 simply states that the solution q'_L

[2]Throughout this paper, we assume the distance function is L_1 (i.e., Manhattan) norm: $\Delta(s,t) = \sum_{i=0}^{d-1} |s[i] - t[i]|$; so, we will drop Δ from the notation. The ideas presented easily generalize to any weighted L_p norm.

[3]Despite the similar name, a reverse (dynamic) skyline query is quite different from a skyline query, focusing on dynamic, spatial proximity and an inversion of the skyline problem.



(a) Points in the plane with close dominance relationships indicated by arrows

(b) In this Sky-not query, p is dominated by three points in the constraint region

(c) By solving the points that closely dominate p, s is also solved transitively

Figure 2: An example of *close dominance* and how it relates to Sky-not queries

must be placed such that it does not dominate any point s that dominates p; otherwise, s will still satisfy the constraints, still dominating p, and thus p will not be in the skyline. Together, these propositions suggest that the solution will lie somewhere inside (q_L, p).

This hyper-rectangular solution space contains infinitely many points. Lemma 3 discretizes the search space. Specifically, it states that the value of q'_L on each dimension i is either the same as the original lower constraint, $q_L[i]$ or as one of the data points, $s[i], s \in \mathcal{S}$. Thus, there is only (an exponential number of) finite possible solutions.

Lemma 3 (Sky-not Discretization)**.**
Let $q'_L = \mathrm{SN}(\mathcal{S}_{(q_L,q_U)}, p, (q_L, q_U))$. *Then,*

$$\forall i \in [0,d) \exists s \in \left(\mathcal{S}_{(q_L,q_U)} \bigcup \{q_L\}\right) \text{ s.t. } q'_L[i] = s[i].$$

The basic idea of the proof is that if there were a solution that was not composed of existing values, then we could find a better one that was.

Proof. We prove this by contradiction. Let q'_L be a solution such that $\exists j \in [0,d) \forall s \in \left(\mathcal{S}_{(q_L,q_U)} \bigcup \{q_L\}\right), q'_L[i] \neq s[i]$. Let $\mathcal{S}_<$ be the points $s \in \mathcal{S}_{(q_L,q_U)} \cup \{q_L\}$ with $s[j] < q'_L[j]$. Let $\overline{s}$ be the point in $\mathcal{S}_<$ with the highest $s[j]$ value. Then, construct a point q''_L such that $q''_L[i] = q'_L[i]$ if $i = j$ and $q''_L[j] = \overline{s}[j]$. Then, q''_L is closer to q_L than is q'_L. Moreover, $\forall s \notin \mathcal{S}_<, \exists i \neq j$ s.t. $s[i] < q'_L[i] = q''_L[i]$, because $q'_L[i] \not\preceq s$ and $q'_L[j] < s[j]$. Also, $\forall s \in \mathcal{S}_<$, it is still the case that $q''_L \not\preceq s$, because $s[j] \leq q''_L[j]$. Lastly, $q''_L \in (q_L, q_U)$, since $q'_L \in (q_L, q_U)$, and they differ only on the j'th attribute, for which $q''_L[j] = q_L[j]$. Therefore, $\forall s \in \mathcal{S}_{(q_L,q_U)}, q''_L[s] \not\preceq s$ and $p \in \mathrm{SKY}(\mathcal{S}, [q''_L, q_U])$. $\square$

Lemma 3 discretized the solution space based on the points $s \in \mathcal{S}$. Lemma 4 reduces the solution space even further by showing that only some of the points that dominate p also influence the solution. For this lemma, we borrow the concept of *close dominance* (Definition 6) from literature: if s dominates t, then it also closely dominates t if there is no other point "in between them":

Definition 6 (Close dominance [22])**.**
Given points $s, t \in \mathcal{S}$, we say that s *closely dominates* t, denoted $s \preccurlyeq t$, iff $s \preceq t \wedge \nexists u \in \mathcal{S} : s \prec u \preceq t$.

Figure 2a illustrates the close dominance relationship. Notice that, although $s \prec\!\!\prec u$, $s \not\preccurlyeq u$, because the dominance can be ascertained transitively through t. Lemma 4 states that it is only the values of points that closely dominate p that might be used in the Sky-not query solution. In Figure 2b, s need not be considered, because $s \not\preccurlyeq p$.

Lemma 4 (Close dominance is sufficient)**.**
Let $C = \{s \in \mathcal{S} : s \preccurlyeq p\}$ *be the set of points in* $\mathcal{S}_{(q_L,q_U)}$ *that closely dominate* p*. Then,*

$$\mathrm{SN}(C, p, (q_L, q_U)) = \mathrm{SN}(\mathcal{S}, p, (q_L, q_U)).$$

The basic idea behind the proof is that by handling all the points in C, one handles all points in $\mathcal{S}_{(q_L,q_U)}$ by means of transitivity.

Proof. We show that finding a point q'_L that does not dominate any point in C is both necessary and sufficient for upgrading p to the skyline.

Necessary: Note that $t \in C \implies t \in \mathcal{S}_{(q_L,q_U)} \wedge t \prec p$, by construction. So, by Proposition 2, $q'_L \not\preceq t$ (i.e., they necessarily fall outside the constraints).

Sufficient: Consider any point $s \in C \setminus \mathcal{S}_{(q_L,q_U)}$. Then, $\exists t \in C$ s.t. $\forall i \in [0,d), s[i] \leq t[i]$, because $s \preceq t$. But, because $t \in C \implies q'_L \not\preceq t$, then $\exists j \in [0,d)$ s.t. $t[i] < q'_L[i]$. By transitivity, $s[i] < q'_L[i]$; so, $q'_L \not\preceq s$. $\square$

The algorithmic consequence of Lemma 4 is that we can first find the much smaller set C and then execute whichever algorithm on C instead of $\mathcal{S}_{(q_L,q_U)}$ for an immediate improvement in efficiency. In Figure 2c, it is sufficient to solve the two points that closely dominate p, because solving t transitively implies that s is also solved.

4.2 Duality

Finally, we show an interesting result, that the Sky-not query is a dual form of a generalization of the SkyDist problem (Definition 7) in literature. The SkyDist problem is to find the cheapest way to modify p so that it will be in the skyline. That is to say, the SkyDist problem changes the data, rather than the query, to upgrade p. We redefine it below with the addition of constraints, which are needed for the duality result.

Definition 7 (Skyline Distance Problem [14]).
Given $\mathcal{S}, p, (q_{\mathrm{L}}, q_{\mathrm{U}})$, find the point $p' \in (q_{\mathrm{L}}, q_{\mathrm{U}})$ closest to p such that $p' \in \mathrm{SKY}(\mathcal{S} \cup \{p'\}, (q_{\mathrm{L}}, q_{\mathrm{U}}))$.

We show that the SkyDist and Sky-not queries are dual problems of each other. First, define a transform function $f(x) = M - x$, overloaded for points and sets, $f(p) = \hat{p} : \hat{p}[i] = f(p[i])$ and $f(\mathcal{S}) = \{f(s) : s \in \mathcal{S}\}$ by applying the function to each value of a point and each point of a set. Let $p' = \mathrm{SD}(\mathcal{S}, p, (q_{\mathrm{L}}, q_{\mathrm{U}}))$ be an instance and solution to a SkyDist problem. Then we have Theorem 5, that the Sky-not query result on the transformed data is exactly the transformation of the SkyDist result on the original data:

Theorem 5 (SkyDist-SkyNot Duality).
$\mathrm{SN}(f(\mathcal{S}), f(p), [f(q_{\mathrm{U}}), f(q_{\mathrm{L}})]) = f(\mathrm{SD}(\mathcal{S}, p, (q_{\mathrm{L}}, q_{\mathrm{U}})))$.

Proof. To show that the solutions are equivalent, we show first that distance is preserved by the transformation function; so that the optimality of any solution is consistent. We then show that the transform of a point that solves the Sky-Not instance solves the SkyDist instance, and vice versa.

Distance-preserving: Let s, t be points. Then:

$$\begin{aligned}\Delta(s,t) &= \sum_{i=0}^{d-1} |s[i] - t[i]| = \sum_{i=0}^{d-1} |(M - s[i]) - (M - t[i])| \\ &= \sum_{i=0}^{d-1} |f(s[i]) - f(t[i])| = \Delta(f(s), f(t)).\end{aligned}$$

This holds in the opposite direction, because $f(f(s)) = s$.

Mutually solving:

$$\begin{aligned} q &= \mathrm{SN}(f(\mathcal{S}), f(p), (f(q_{\mathrm{U}}), f(q_{\mathrm{L}}))) \\ &\implies f(q_{\mathrm{U}}) \prec q \prec f(q_{\mathrm{L}}), \forall s \in f(\mathcal{S}), q \not\prec s \\ &\implies \exists i \in [0,d) : s[i] < q[i] \\ &\implies \exists i \in [0,d) : f(q[i]) < f(s[i]) \implies f(s) \not\prec f(q) \\ &\implies q_{\mathrm{L}} \prec f(q) \prec q_{\mathrm{U}}, \forall s \in \mathcal{S}, s \not\prec f(q). \\ f(p') &= f(\mathrm{SD}(\mathcal{S}, p, (q_{\mathrm{L}}, q_{\mathrm{U}}))) \implies \forall s \in \mathcal{S}, s \not\prec p' \\ &\implies f(p') \not\prec f(s) \implies \forall s \in f(\mathcal{S}), f(p') \not\prec s. \end{aligned}$$

□

In fact, the above holds for any order-inverting and distance-preserving transformation function.

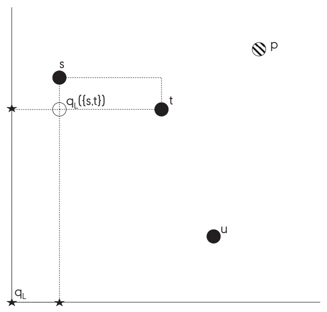
The signficance of this result is that techniques developed for SkyDist and Sky-not queries are mutually exchangeable by applying the transform. However, the addition of constraints enables new analytical approaches that lead to more efficient computation, as we will show in our experimental evaluation, where we apply the duality transform to existing SkyDist techniques (Section 6).

5. ANSWERING SKY-NOT QUERIES

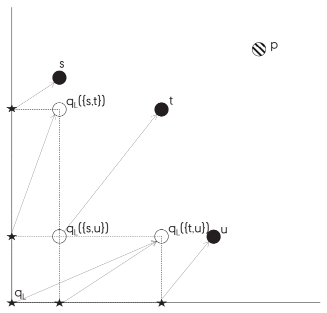
In this section, we present two algorithms for solving Sky-not queries, based on the insight from the previous section.

5.1 Bounding Rectangle Algorithm

Here, we introduce the Bounding Rectangle Algorithm (BRA). We begin with theoretical analysis to deduce a finite set of candidate positions to which q_{L} can be moved in order to position p in the skyline (Section 5.1.1). We then build an algorithm to efficiently calculate those positions and find the optimal solution from within them (Section 5.1.2).



(a) The creation of $q_{\mathrm{L}}(\{s,t\})$ with the minimum value from $\{s,t\}$ on each attribute and of rectangle $(q_{\mathrm{L}}, q_{\mathrm{L}}(\{s,t\}))$. Stars represent the 2^d projections of the rectangle.



(b) All three possible combinations of two points are shown with their resultant rectangles. Arrows represent strict dominance relationships, the origin of which are points that are not valid solutions.

Figure 3: Bounding rectangles and their application in BRA.

5.1.1 Bounding Rectangles and Candidate Positions

Recall from Lemma 3 that solutions will take values from points in the dataset and from Lemma 4 that the set C contains all the points that need to be considered. We formalize that set with Corollary 6 below. We then show that this set can be further pruned by taking into account how these points relate to each other (Theorem 7).

First, however, we need to introduce a little more notation for use in this subsection. Let $\mathcal{P}_{\leq d}(C)$ be the subsets of C with at most d elements, and let $T \in \mathcal{P}_{\leq d}C$ be such a subset. Furthermore, let $q_{\mathrm{L}}(T)$ be the corner of the minimum bounding rectangle of T that is closest to q_{L} (the bottom left corner in two dimensions). Notice that $\mathrm{R}_T = (q_{\mathrm{L}}, q_{\mathrm{L}}(T))$ is itself a (possibly degenerate) hyper-rectangle. Finally, denote by $\llcorner(\mathrm{R}_T)$ the 2^d corners of $(q_{\mathrm{L}}, q_{\mathrm{L}}(T))$.

Recall that a *minimum bounding rectangle* of a set of points T is the unique smallest hyper-rectangle that contains all points in T. The lowermost corner (i.e., the one closest to q_{L}) is the one containing the smallest value on each attribute of any point in T. It represents the best possible combination of values from points in T. Sets with more than d points are not interesting, because we only combine values for up to d dimensions.

Figure 3a illustrates these concepts. The set $C = \{s,t,u\}$ has $\mathcal{P}_{\leq 2}C = \{\{\}, \{s\}, \{t\}, \{u\}, \{s,t\}, \{s,u\}, \{t,u\}\}$. Let $T = \{s,t\}$. The minimum bounding rectangle of T is the dashed rectangle to the top-right in the figure, and its $q_{\mathrm{L}}(T)$ is shown as a hollow circle. The dashed rectangle to the lower-left is $(q_{\mathrm{L}}, q_{\mathrm{L}}(T))$. The 2^d corners are marked with stars.

Each of the 2^d corners of hyper-rectangle $(q_{\mathrm{L}}, q_{\mathrm{L}}(T))$ has a unique set of attributes with values equal to those of q_{L} and the others equal to those of $q_{\mathrm{L}}(T)$. They correspond to solutions with fewer than all values changed. So, Corollary 6 states that if we take all subsets T of up to d points from C and consider all the corners on the hyper-rectangle traced from q_{L} to $q_{\mathrm{L}}(T)$, we will include all possible combinations of all points in C, and therefore also have the optimal solution somewhere in the (finite) set.

Corollary 6 (BRA search space)**.**
$q'_{\mathrm{L}} = \mathrm{SN}(\mathcal{S}, p, (q_{\mathrm{L}}, q_{\mathrm{U}})) \in \bigcup_{T \in \mathcal{P}_{\leq d}(C)} \llcorner(\mathrm{R}_T)$.

Proof. This follows directly from Lemmata 3 and 6, because all possible combinations of all points in C are contained in $\bigcup_{T \in \mathcal{P}_{\leq d}(C)} \llcorner(\mathrm{R}_T)$. ☐

Corollary 6 includes all possible solution points, but also many more. Theorem 7 gives the key idea for the BRA, that it is only those points in the set that do not dominate any others in the set that are possibilities. The property of not strictly dominating any other corner points is both necessary and sufficient for indicating that any given corner point correctly positions p in the skyline.

Theorem 7 (Central BRA postulate)**.**
q solves $\mathrm{SN}(\mathcal{S}, p, (q_{\mathrm{L}}, q_{\mathrm{U}}))$ *iff* $\forall \llcorner \in \bigcup_{T \in \mathcal{P}_{\leq d}(C)} \llcorner(\mathrm{R}_T), q \not\prec\!\!\prec \llcorner$.

The basic idea behind the proof is that it is certainly sufficient, because every point $c \in C$ is a corner point for some rectangle; so, not dominating any corner points for any rectangles implies that no point in c is dominated either. It is necessary because any dominated point will produce some rectangle with a corner point that is also dominated.

Proof. **Sufficient**:

$$\begin{aligned} & \forall \llcorner \in \bigcup_{T \in \mathcal{P}_{\leq d}(C)} \llcorner(\mathrm{R}_T), q \not\prec\!\!\prec \llcorner \\ \implies \; & \forall \llcorner \in \bigcup_{T \in \mathcal{P}_{\leq 1}(C)} \llcorner(\mathrm{R}_T), q \not\prec\!\!\prec \llcorner \\ \implies \; & \forall \llcorner \in \bigcup_{t \in C} \llcorner([q_{\mathrm{L}}, t]), q \not\prec\!\!\prec \llcorner \implies \forall t \in C, q \not\prec\!\!\prec t. \end{aligned}$$

Necessary: Assume for the sake of contradiction that $\exists \llcorner \in \bigcup_{T \in \mathcal{P}_{\leq d}(C)} \llcorner(\mathrm{R}_T)$ s.t. $q \prec\!\!\prec \llcorner$. Let T denote the subset of C that produced the hyper-rectange with that corner. Then, $\exists t \in T : q \prec\!\!\prec t \implies \exists s \in C : q \prec\!\!\prec s$ and, thus, q cannot be a valid solution of $\mathrm{SN}(\mathcal{S}, p, (q_{\mathrm{L}}, q_{\mathrm{U}}))$. ☐

The concept behind Theorem 7 is illustrated in Figure 3b. Three of the hyper-rectangles are shown, along with all their corner points. Arrows depict that the originating corner strictly dominates the destination corner. One can verify in the figure that all positions with arrows dominate some point in C, whereas all positions without arrows (namely $q_{\mathrm{L}}(\{s, t\})$ and $q_{\mathrm{L}}(\{t, u\})$) do not.

5.1.2 *Algorithm description*

Theorem 7 suggests an elegant way to adapt existing skyline algorithms to solve a Sky-not query, since the candidate solutions are the points in $X = \bigcup_{T \in \mathcal{P}_{\leq d}(C)} \llcorner(\mathrm{R}_T)$, and the optimal solution is the point $x \in X$ closest to q_{L}. So, we adapt the Block-Nested-Loop (BNL) [3] skyline algorithm, using a window sorted by proximity to q_{L}, to produce the set X. The head of the window once all points have been processed is the optimal solution, since Corollary 6 states that all possible solutions are in that set.

Algorithm 1 Bounding Rectangle Algorithm (BRA)

```
Input: $\mathcal{S}, p, q_{\mathrm{L}}, q_{\mathrm{U}}$
Output: $q'_{\mathrm{L}}$, the optimal Sky-not solution on $(\mathcal{S}, p, (q_{\mathrm{L}}, q_{\mathrm{U}}))$
 1: Create empty set $C$
 2: for all $s \in \mathcal{S}$ do
 3:   if $q_{\mathrm{L}} \preceq s \preceq p \preceq q_{\mathrm{U}}$ then
 4:     for all $c \in C$ do
 5:       if $s \prec c$ then
 6:         add← false; break
 7:       else if $c \prec s$ then
 8:         $C \leftarrow C \setminus \{c\}$
 9:     if add then
10:       $C \leftarrow C \bigcup \{s\}$
11: Create queue W sorted by ascending proximity to $q_{\mathrm{L}}$
12: for all $\llcorner \in \bigcup_{T \in \mathcal{P}_{\leq d}(C)} \llcorner(\mathrm{R}_T)$ do
13:   for all $p \in$ W do
14:     if $\llcorner \prec\!\!\prec p$ then
15:       add←false; break
16:     else if $p \prec\!\!\prec \llcorner$ then
17:       W ← W \ $\{p\}$
18:   if add then
19:     W ← W $\bigcup \{\llcorner\}$
20: Return W[0]
```

Specifically, Algorithm 1 first computes the set C (Lines $1 - 10$) and then iterates through all corner points to find those that are valid solutions (Lines $11 - 19$). By storing the valid solutions in a queue that is sorted by proximity to q_{L}, the head of the list is the optimal solution (Line 20).

Both steps follow the same control flow. We go through every point $s, \llcorner$ and compare it to every other point c, p in our current solution set. If s is dominated by c, then c is removed from the current solution set. If s does not dominate any c, then it is added. The difference with $\llcorner, p$ is that we require strict dominance.

The running time of the algorithm thus depends on the maximum size that the window W becomes. We know the final solution is correct and optimal on account of Theorem 7.

5.2 Prioritized Recursion Algorithm

In this section, we introduce our Prioritized Recursion Algorithm (PrioReA), which uses a few theoretical conclusions to discover good solutions very quickly, and use it to prune the majority of the search space.

5.2.1 *Algorithmic Foundations*

The objective in this section is to build towards a sound recursive formulation of the problem and a series of theoretical pruning rules that can be used to limit the search space. We can then describe a recursive algorithm, based on that formulation, which uses the pruning rules to pursue the optimal solution dramatically faster.

We begin with two lemmata that give rise to the pruning rules and recursion. First, we note that the problem is monotonic in the sense that solutions on subsets of points are necessarily at least as good as those on supersets. Specifically, Lemma 8 states that if one adds points to $\mathcal{S}$, the cost of the solution cannot decrease. The key observation is that adding more points to an input set forces a solution to be farther from q_L if it is not to strictly dominate any of the points in either the subset nor the superset.

Lemma 8 (Monotonically increasing cost)**.**
Let $q'_\mathrm{L} = \mathrm{SN}(\mathcal{S}, p, (q_\mathrm{L}, q_\mathrm{U}))$ *and* $q''_\mathrm{L} = \mathrm{SN}(\mathcal{S}', p, (q_\mathrm{L}, q_\mathrm{U}))$. *Then,* $\mathcal{S} \subset \mathcal{S}' \implies \Delta(q_\mathrm{L}, q'_\mathrm{L}) \leq \Delta(q_\mathrm{L}, q''_\mathrm{L})$.

Proof. q'_L is the closest point to q_L that dominates all points in $\mathcal{S}$. q''_L must also dominate all points in $\mathcal{S}$, since $\mathcal{S} \subset \mathcal{S}'$; so, it cannot be closer to q_L than q'_L is. □

The second lemma defines a lower bound on the solution cost. Specifically, Lemma 9 states that, if one discovers the point s whose smallest attribute value (relative to q_L's) is farthest from q_L's value on that attribute, this difference lower bounds the cost of the optimal solution. Here, the observation is that this value is the smallest value that could conceivably guarantee that no point is strictly dominated.

Lemma 9 (Lower bound on solution cost)**.**
$\max_{s \in \mathcal{S}} \min_{i \in [0,d)} (s[i] - q_\mathrm{L}[i]) \leq \mathrm{SN}(\mathcal{S}, p, (q_\mathrm{L}, q_\mathrm{U}))$.

Proof. Since $\forall s \in \mathcal{S}, q'_\mathrm{L} \not\prec\!\!\prec s$, q'_L must be larger or equal to at least one value of every point. □

Lemma 8 is valuable for pruning the search space, because it indicates that if we find a recursive call to be unpromising, then all recursive calls using a superset of those points will be likewise unpromising. Lemma 9 is useful because it allows estimating the optimal solution that will be returned by a recursive call without actually calculating it. Thus, we can often use the estimate to determine that a recursive call cannot possibly produce a better solution than what we have already seen.

This brings us to the main theorem for this section, which formulates the problem recursively. Specifically, Theorem 10 states that if we partition a set of points C based on whether they have a value $> x$ on dimension i, then the solution q'_L with $q'_\mathrm{L}[i] = x$ that is closest to q_L is exactly the difference between $q'_\mathrm{L}[i]$ and $q_\mathrm{L}[i]$ plus the cost of the best possible solution with $q'_\mathrm{L}[i] = q_\mathrm{L}[i]$ on the higher-valued partition.

Theorem 10 (Basis for recursion)**.**
Let $C_{i>x} = \{s \in C : s[i] > x\}$ *and* $Q_{i>x} = \{q \in [q_\mathrm{L}, p] : \forall s \in C_{i>x}, q \not\prec\!\!\prec s\}$. *Then, for* $x \geq q_\mathrm{L}[i]$*:*

$$\begin{aligned} & \Delta(q_\mathrm{L}, \mathit{argmin}_{q \in Q_{i>q_\mathrm{L}[i]}: q[i]=x} \Delta(q_\mathrm{L}, q)) \\ = \quad & \Delta(q_\mathrm{L}, \mathit{argmin}_{q' \in Q_{i>x}: q'[i]=q_\mathrm{L}[i]} \Delta(q_\mathrm{L}, q')) + (x - q_\mathrm{L}[i]). \end{aligned}$$

Proof. Note that for a point q with $q[i] = x$, $\nexists s \in C$ with $s[i] \leq x$ such that $q \prec\!\!\prec s$. All other points, those in $C_{i>x}$, must have a lower value than q on some other attribute; i.e., must be dominated by the projection of q onto the point q' where $q'[i] = q_\mathrm{L}[i]$ (i.e., is not changed from the original constraint). So, $\Delta(q_\mathrm{L}, q)$ is exactly the distance of the projection q' from q_L plus the distance from q to q', since all distances are positive in all directions. □

Finally, we note the three pruning rules that are straightforward consequences of the earlier lemmata in this section. In particular, Corollary 11 states that if we have already seen a point whose cost (i.e., distance to q_L) is less than the sum of the distance of a given point from q_L and the lower bound on the recursive call, then, independent of the recursive call, the given point cannot produce a better solution than what has already been seen. This comes directly from Lemma 9 and Theorem 10.

Algorithm 2 Prioritized Recursion Algorithm (PrioReA)

```
Input: $C, p, q_\mathrm{L}, \mathcal{D}$
Output: $q'_\mathrm{L}$, the optimal Sky-not solution on $(\mathcal{S}, p, (q_\mathrm{L}, q_\mathrm{U}))$
 1: if $C = \emptyset$ then
 2:     Return $q_\mathrm{L}$
 3: best← $p$
 4: Sort $d \in \mathcal{D}$ by $p[d] - q_\mathrm{L}[d]$, ascending
 5: for all $d \in \mathcal{D}$ do
 6:     Sort $s \in \mathcal{S}$ by $s[d] - q_\mathrm{L}[d]$, descending
 7:     maxmin ← 0
 8:     for all $s \in C$ do
 9:         if $s[d] - q_\mathrm{L}[d] + \mathrm{maxmin} < \Delta(q_\mathrm{L}, \mathrm{best})$ then
10:             rec←PrioReA($C_{[0,\ldots,s-1]}, \mathcal{D} \setminus \{d\}, q_\mathrm{L}, p$)
11:             if $\Delta(q_\mathrm{L}, \mathrm{rec}) + s[d] - q_\mathrm{L}[d] < \Delta(q_\mathrm{L}, \mathrm{best})$ then
12:                 best←rec, best[$d$] ← $s[d]$
13:             else if $\Delta(q_\mathrm{L}, \mathrm{rec}) \geq \Delta(q_\mathrm{L}, \mathrm{best})$ then
14:                 Break {Pruning rule (2).}
15:         else if maxmin ≥best then
16:             Break {Pruning rule (2').}
17:         else
18:             Do nothing {Pruning rule (1).}
19:         if maxmin < $\min_{d' \in \mathcal{D}} s[d'] - q_\mathrm{L}[d']$ then
20:             maxmin ← $\min_{d' \in \mathcal{D}} s[d'] - q_\mathrm{L}[d']$
21: Return best
```

Corollary 11 (Pruning Rule (1))**.**
If $\exists q \in (q_\mathrm{L}, p), i \in [0, d), x \geq q_\mathrm{L}[i]$ *s.t.*
$\Delta(q_\mathrm{L}, q) \leq (x - q_\mathrm{L}[i]) + \max_{s \in C_{i>x}} \min_{i \in [0,d)} (s[i] - q_\mathrm{L}[i]))$
then $\forall q' \in (q_\mathrm{L}, p) : q'[i] = x, \Delta(q_\mathrm{L}, q) \leq \Delta(q_\mathrm{L}, q')$.

The second pruning rule, Corollary 12, states that if we have already seen a point whose cost is less than the cost of the best solution returned by a recursive call on C', then we need never consider any recursive calls on supersets of C'. This comes directly from Lemma 8.

Corollary 12 (Pruning Rule (2))**.**
If $\exists q \in (q_\mathrm{L}, p), i \in [0, d), x \geq q_\mathrm{L}[i]$ *s.t.*
$\Delta(q_\mathrm{L}, q) \leq \Delta(q_\mathrm{L}, \mathit{argmin}_{q' \in Q_{i>x}: q'[i]=q_\mathrm{L}[i]} \Delta(q_\mathrm{L}, q'))$ *then*
$\forall x' \leq x, \Delta(q_\mathrm{L}, q) \leq \Delta(q_\mathrm{L}, \mathit{argmin}_{q' \in Q_{i>x'}: q'[i]=q_\mathrm{L}[i]} \Delta(q_\mathrm{L}, q'))$.

Finally, a variant of the second pruning rule, Corollary 13, states that if we have already seen a point with cost less than the lower bound estimate of the recursive call, we can also safely dismiss all recursive calls on supersets.

Corollary 13 (Pruning Rule (2'))**.**
If $\exists q$ *s.t.* $\Delta(q_\mathrm{L}, q) < \max_{s \in C} \min_{i \in [0,d)} (s[i] - q_\mathrm{L}[i])$ *then,*
$\forall C', C \subseteq C'$, *$q$ is the best possible solution.*

5.2.2 Algorithm Description

Algorithm 2 describes the PrioReA algorithm. At a high level, it is a recursive algorithm, based on Theorem 10, that selects a dimension d and fixes a value x for that dimension. Any point s with $s[d] \leq x$ is clearly not strictly dominated. With all the other points $s' \in C, s'[d] > x$, and the remaining dimensions $d' \neq d$, we recurse to find an optimal solution. The combination of the result from the recursion with the value x on dimension d is $\text{argmin}_{q' \in Q: q'[i]=x} \Delta(q_{\text{L}}, q')$, the best possible solution with value x on dimension d.

To set the value x on a recursive call with points C', we sort the points in $c \in C'$ by descending $c[d]$ value (Line 6) and iterate the sorted list (Line 8). We know from the discretization lemma, Lemma 3, that these are the only values that need to be considered. By iterating in descending order, we prioritize recursive calls with smaller inputs: it is the set of all points preceding the current one that still need to be resolved on the recursive call.

If we first compute C as in Lines $1-10$ of Algorithm 1, and then we iterate through all dimensions (Line 5) and for all possible values to which each of those dimensions could be set (Line 8), we are guaranteed to find the optimal solution, because we will have covered the entire search space of BRA that was defined in Corollary 6, with performance $\mathcal{O}(n^d)$.

However, PrioReA uses Corollaries 11-13 to avoid the majority of that search space. We maintain track of the *best* solution globally seen, the point q in the corollaries. Then, whenever the conditions of the corollaries are met (Lines 13, 15, or 17), we can avoid the recursive call (Line 18) or even break the loop entirely (Lines 14 or 16).

Furthermore, we prioritize the recursive calls exactly to push the *best* seen point, q, as close to q_{L} as early as possible. Specifically, we always choose next the dimension wherein p is closest to q_{L}, because this increases the likelihood of finding a good value x on that dimension that is also close to q_{L}. We always choose points closest to p first, rather than q_{L}, so that the recursive calls have fewer points and we need to change fewer dimensions to reach a solution.

The effectiveness both of the pruning rules and of the prioritizations we evaluate next, in Section 6.

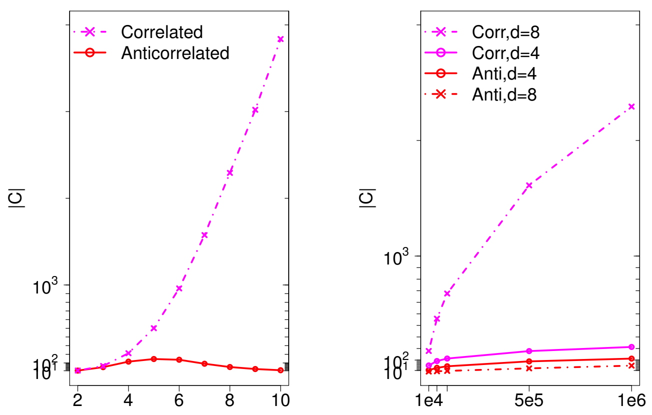
6. EXPERIMENTAL EVALUATION

In this section, we provide an extensive experimental evaluation of the contributions made thus far. We describe the basic, common setup for the experiments in Section 6.1. We evaluate the impact of Section 4.1 in Section 6.2 by measuring how large is the set C of points that closely dominate p, as a function of the input parametres. In Section 6.3, we compare the query performance of our two algorithms from Section 5 against adaptations using Theorem 5 of state-of-the-art algorithms for the SkyDist problem. Finally, in Section 6.4, we investigate our recursive algorithm in more depth, particularly the efficacy of its pruning rules and the average depth of the recursion.

6.1 Experimental Setup

Algorithms: We implement four algorithms in C++ for comparison. This includes both BRA and PrioReA from Section 5. We also adapt and implement the Sort-Projection (SORT_PROJ) and Space-Partition (SPACE_PART) methods of Huang et al. [12], since these were shown to be the most efficient SkyDist algorithms [12]. All four implementations start from the same reduced input set, C, based on the theoretical analysis in Section 4.

Environment: All experiments are run on a commodity machine with an i7-2700 quad-core processor clocked at 3.4 GHz, 16 GB of memory, and Ubuntu on kernel version 3.13.0. The code is compiled using the GNU C++ compiler version 4.8.2 with full optimization. Since BRA is embarassingly parallel (and the slowest running), we run it on 8 threads (hyperthreading is enabled). All other algorithms are single-threaded.[4]



(a) $|C|$ vs. d ($|\mathcal{S}| = 10^6$). (b) $|C|$ vs. $|\mathcal{S}|$ ($d = 4, 8$).

Figure 4: Variation of the size of the reduced input, C, relative to the full input, $\mathcal{S}$.

Datasets: We generate random datasets using the data generator standard to skyline research [3] to produce normalized datasets of *anticorrelated* (A) and *correlated* (C) distributions. The selection of dataset cardinality (n) and dimensionality (d), query constraints ($q_{\text{L}}, q_{\text{U}}$), and query points ($p$) varies according to the objective of each sub-study.

6.2 Regarding close dominance and $|C|$

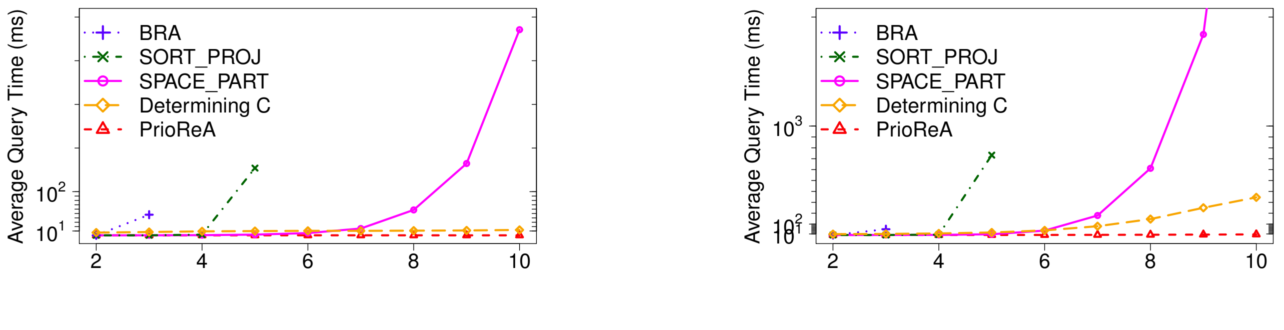
Experiment description: By Lemma 4, it is not the size of the input dataset that governs performance, but the size of the set C, the points that closely dominate p. We begin by empirically gauging the impact on the input size it has to apply this lemma.

We run 10^4 random trials for each configuration and report the average. For each trial, we pick a point $p \in \mathcal{S}$ uniformly at random. We then select each lower constraint, $q_{\text{L}}[i]$, uniformly from the data range, $[0, p[i]])$, independently for each attribute. Since q_{U} does not influence the set C, we set it to the maximum value (i.e., 1) on each attribute. With this setup, we then compute the set C using Lines $1-10$ of Algorithm 1 and record the number of points, $|C|$.

Results and discussion: The results are reported in Figure 4. In Figure 4a, we vary data dimensionality (d) from $2-10$ in increments of 1 and hold the data cardinality ($|\mathcal{S}|$) fixed at one million points. The pink line with x's shows the results for correlated data, and the orange line with o's, for anticorrelated data. Note that the y-axis, representing $|C|$, is logarithmic.

The results in this plot are very surprising, because correlated data exhibits more challenging behaviour than the anticorrelated data. In typical skyline experiments, anticorrelated data produces larger skylines (output). This slows performance because performance is typically dependent on the size of the output. Similarly, increases in d also hinder performance, because they also increase the size of the output. Here, however, we see that the reduced input set for

[4]Parallelizing the slowest running algorithm allows us to scale up the experiments without compromising fairness among the competitive algorithms.



(a) Anticorrelated

(b) Correlated

Figure 5: Execution time of the algorithms and the computation of C as a function of input parametres ($|\mathcal{S}| = 10^6$).

Sky-not queries is larger on correlated data, and grows exponentially with d. On anticorrelated data, it peaks around $d = 6$, then decreases with subsequent increases in d.

This counter-intuitive behaviour can be understood clearly by considering the extra requirement imposed by close dominance (Definition 6): that $c \in C$ iff $\nexists c' \in \mathcal{S}$ s.t. $c \prec c' \prec p$. On correlated data, it is likely that, for a randomly chosen point p, many other points dominate it. However, there are also many dominance relationships among them. As d increases, the points that dominate p become incomparable to each other. For points $c \prec c'$ that both dominate p, only $c \in C$. However, if $c \prec\succ c'$, then $c' \in C$ as well.

Considering anticorrelated data, on the other hand, there are much fewer points that originally dominate p. So, for low-dimensional increases to d, we observe the same trend as with correlated data, but less pronounced. However, by $d = 6$, this effect has been saturated, and most points that dominate p are incomparable to each other. For subsequent increases in d, many points in C that dominate p become incomparable to it; so, the size of the set shrinks.

Figure 4b, on the other hand, shows behaviour with increases data cardinality ($|\mathcal{S}|$). Because different values of d exhibited quite different results, we plot twice as many curves here: two for $d = 4$ and two for $d = 8$ (either side of the peak for the anticorrelated data). The pink lines are, again, correlated data; the orange, anticorrelated. The x's correspond to $d = 8$ and the o's, $d = 4$.

Here, we see easily predicted behaviour. By increasing the number of points in $\mathcal{S}$, we consequently increase the number of points in C. For anticorrelated data and for $d = 4$ on correlated data, the relationship is roughly linear, but the savings offered by Lemma 4 is dramatic. Only $1/10^4$ of the points need to be processed. All possible solutions involving values from any of the other 9999/10000 of the data points can be discarded early in preprocessing by all algorithms.

The exceptional case is the higher-dimensional, correlated data. This again grows steeply in accordance with Figure 4a. Of the additional points, many dominate p because the data is correlated, but many also are incomparable to each other because of the higher dimensionality.

In summary, this study shows that Lemma 4 can dramatically reduce the input, and thus also the search space. Counter-intuitively, it also demonstrates that for Sky-not queries, correlated data is much more challenging than anticorrelated data, with the former becoming exponentially more challenging with increases in $d > 6$ and the latter becoming easier under the same conditions.

6.3 On the scalability of the algorithms

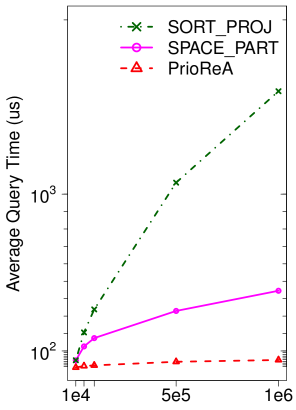
Experiment description: We next compare the four algorithms in terms of execution time, using the same experimental configurations as in Section 6.2. In contrast to the query generation methodology of [12], which first chooses m skyline points, and then generates a virtual query point that is dominated by all of them, our methodology ensures that query points still come from the original underlying distribution. So, we expect to observe performance following the trends illustrated in Figure 4. For readability, we separate the cardinality plots using different dimensionality.

Results and discussion: Figure 5 shows performance of the algorithms with respect to d. We also include the time taken to compute the set C, which is not included in the time for any of the four algorithms. Generally speaking, it is relatively efficient compared to all the algorithms. The exception, relative to PrioReA, on correlated data for higher d, is quite interesting. Recall that C is computed BNL-style with a window of current points to which each candidate is compared. Much like anticorrelated data slows skyline computation in BNL with increasing d, the correlated data slows the generation of C with increasing d.

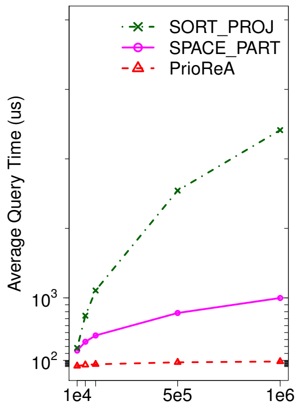
Figure 5a gives performance on anticorrelated data, and Figure 5b, correlated. A first observation is the different scales on the y-axis. As expected from the larger C input set, the correlated data generally takes longer to compute for all algorithms. Another immediate observation is that, excepting PrioReA, all the algorithms deterioate rapidly after a threshold dimensionality. For BRA, this is unfortunately just $d = 3$; so, we exclude it from all subsequent experiments due to its poor scalability. Evidently, the combination of a large window size, `W`, and a still quite large search space, make BRA prohibitively slow.

In agreement with the findings in [12], we observe SPACE_PART to be both faster and more scalable than SORT_PROJ. They both deteriorate rapidly after a threshold dimensionality–SORT_PROJ at $d = 5$ and SPACE_PART at $d = 8$. Interestingly, these thresholds are independent of the data distribution, which suggests that it is not $|C|$, which is decreasing on the anticorrelated data, that is the cause. Rather, it is strictly a function of the dimensionality.

In contrast, PrioReA scales quite gracefully with increasing dimensionality, independent of the data distribution. We investigate why in Section 6.4.

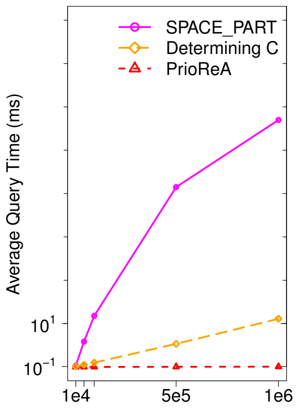


(a) Anticorrelated

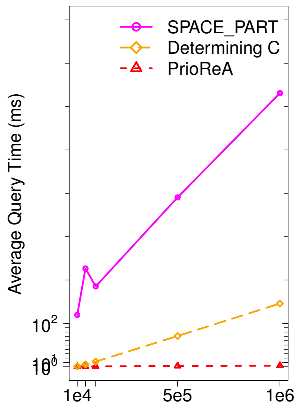


(b) Correlated

Figure 6: Query performance vs. $|\mathcal{S}|$ ($d = 4$).



(a) Anticorrelated



(b) Correlated

Figure 7: Query performance vs. $|\mathcal{S}|$ ($d = 8$).

Next, we consider increasing $|C|$. Figure 6 shows results with $d = 4$ and Figure 7 shows results with $d = 8$. Because BRA did not scale to these dimensionalities, we exclude it from these plots. Similarly, SORT_PROJ did not scale to $d = 8$; so, we instead show the preprocessing time.

With the exclusion of the slower-performing algorithms, we can portray a more granular comparison of the algorithms. We see now that, even at $d = 4$, PrioReA is already an order-of-magnitude faster than the next fastest algorithm. We will investigate this difference in the next subsection. All three algorithms scale gracefully with increases in cardinality, but SORT_PROJ and SPACE_PART have a much larger scaling factor than PrioReA.

In summary, this study shows that PrioReA has much better scalability than the other three algorithms, with respect to both d and $|\mathcal{S}|$. We also see that the preprocessing phase is efficient for the savings in Section 6.2.

6.4 Granular Analysis of Recursion

We saw in Section 6.3 that SPACE_PART substantially outperforms both BRA and SORT_PROJ and that PrioReA outperforms SPACE_PART by an additional order of magnitude. Here, we conduct more granular experiments to explain the performance. Since both SPACE_PART and PrioReA are recursive algorithms, we compare them in Section 6.4.1 based on the number of recursive calls each makes. Then, in Section 6.4.2, we evaluate the success rate of PrioReA's pruning rules from Section 5.2.1.

6.4.1 Number of recursive calls

Experiment description: In order to better understand the discrepancy in performance between SPACE_PART and PrioReA, we study here the average number of recursive calls made by each algorithm. To do this, we insert a global counter variable into the source code of the recursive method for each algorithm. We adopt the same experimental configurations as in the earlier tests.

Results and discussion: Figure 8 gives results of the recursion experiments. Here, the pink lines represent SPACE_PART and the orange lines represent PrioReA. The lines with x's show correlated data and the lines with o's, anticorrelated. The y-axis is the number of times the recursive method is invoked per Sky-not query, plotted on a logarithmic scale.

Figure 8a shows the variation with respect to d with $|\mathcal{S}| = 10^6$. It is worth contrasting this to Figure 5, the query times for the same configurations. On the correlated data, we observe that the number of recursive calls made by SPACE_PART grows dramatically, even on the logarithmic scale. Note that the growth pattern of $|C|$ in Figure 4a has the same inflection point at $d = 4$. Intuitively, the number of recursive calls is increasing with the effective input size, $|C|$.

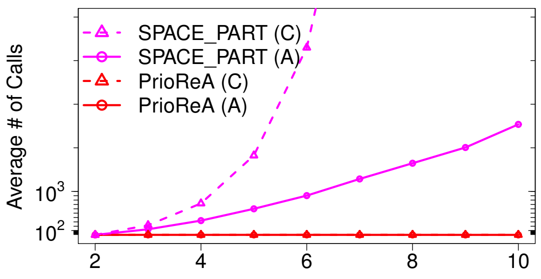
For the anticorrelated data, the number of recursive calls still grows exponentially for SPACE_PART, but more controllably. With each addition of a dimension, the number of partitions created by SPACE_PART doubles, but the occupancy of these partitions shrinks (because the same number of points are distributed among a larger number of partitions).

In contrast, the number of recursive calls made by PrioReA is stable with increasing d, indicating that the condition on Line 9 of Algorithm 2, which tests whether or not to recurse on a subset of dimensions, is not affected strongly by the *number of* dimensions, only by the quality of the solutions yet seen. This effect is distribution-independent.

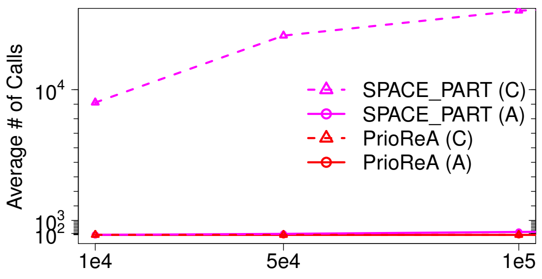
Figure 8b shows the variation with respect to $|\mathcal{S}|$ with $d = 8$. Note that the scale of the y-axis is different from the previous plot. We only show the results for $d = 8$, since they are mirrored, but less pronounced, at $d = 4$. Contrast these results to the query times in Figure 7. Here, we observe that, in contrast, to increases in d, the behaviour of both algorithms is relatively stable. In fact, they almost exactly mirror the trends in Figure 4b, which show $|C|$. With increasing input size, the algorithms both incur proportionately more recursive calls.

Note, importantly, that there is a large deviation, over an order of magnitude, between the number of recursive calls made by SPACE_PART and PrioReA on anticorrelated data. It is difficult to see in the plot because the range of the scale must be very large to fit SPACE_PART on correlated data.

In summary, we see the query time performance of SPACE_PART exhibits the same behaviour as the number of recursive calls, with sharp inflection points as d increases and more stable growth with $|\mathcal{S}|$. For PrioReA, the stable query times are matched by a stable number of recursive calls.

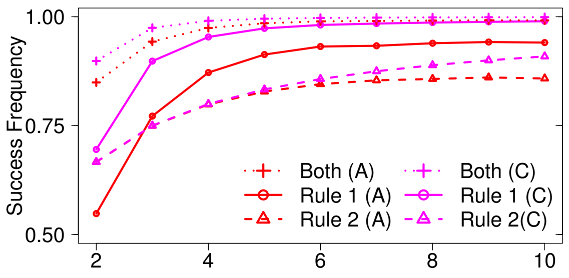


(a) # of recursive calls vs. d ($|\mathcal{S}| = 10^6$).

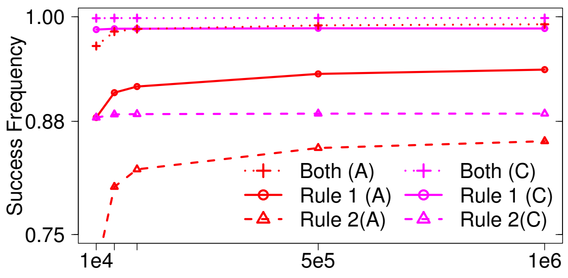


(b) # of recursive calls vs. $|\mathcal{S}|$ ($d = 8$).

Figure 8: Number of recursive calls relative to input configurations.



(a) Success rate of pruning rules vs. d ($|\mathcal{S}| = 10^6$).



(b) Success rate of pruning rules vs. $|\mathcal{S}|$ ($d = 8$).

Figure 9: Success rate of pruning rules relative to input configurations.

6.4.2 Pruning efficacy

Experiment description: Our last experiment investigates why the number of recursive calls for PrioReA is so stable. In particular, because recursive calls are averted by the pruning rules introduced in Section 5.2.1, we evaluate their success rates. We use, again, the same configurations, and add global counters at Lines 12, 14, 16, and 18 of Algorithm 2 to profile the percentage of invocations taking each path through the possible conditions.

Results and discussion: Figure 9 presents the results. Pruning rules 2 and 2′ are combined, since they both break the loops early based on the result of, or estimation of, the quality of the solution on the recursion, respectively. Pruning rule 1, on the other hand, does not break the loop, and is based on an estimate of the cost for a current value, $s[d]$.

Here, the y-axis indicates the fraction of times that the relevant condition evaluates favourably for the given pruning rule. The pink lines represent correlated data and the orange lines, anticorrelated. Rule 1 is depicted with o's and Rule 2, with x's. The combined success rate, $1 - (1 - \text{Rule1}) * (1 - \text{Rule2})$, is computed analytically and plotted.

Figure 9a shows the success rates with respect to d, with $|\mathcal{S}| = 10^6$. Interestingly, as d increases, so does the success rate of both pruning rules. This is the effect of our prioritizing the most promising (i.e., closest) dimensions first, obtaining good solutions very quickly. As the dimensionality increases, so do the number of choices of dimensions to prioritize. Additionally, the reasonably good solutions are found quite quickly, regardless of how many iterations through the loops we need, and can subsequently improve pruning power. As d increases, so too does the number of iterations through the loops on the first recursive call. However, the first few find the good solutions and boost the pruning potential for the increased number of subsequent iterations.

Rule 1 is, for $d > 2$, more effective than Rule 2. However, it is also tested first in Algorithm 2. So, easily pruneable cases are pruned by Rule 1 and never evaluated by Rule 2. All cases evaluated by Rule 2 were missed by Rule 1; so, it is not surprising that the success rate of pruning these cases is slightly lower.

Both pruning rules are more often successful on correlated data than on anticorrelated data. As with increases in d, this is because there are more tests in general (the loop on Line 6 of Algorithm 2 is larger), but a relatively constant number of unsuccessful, early evaluation of the pruning conditions.

By $d = 4$, we see that the combined success rate of the pruning rules is very nearly 100% on both data distributions, which clearly explains why the number of recursive calls observed in Section 6.4.1 was stable with increasing d.

Figure 9b shows the success rates with respect to $|\mathcal{S}|$, with $d = 8$. As before, we omit highly similar results for $d = 4$. Here we see that, aside from initial relatively large jumps in input size, increases in cardinality do not especially affect the pruning rates. At $d = 8$, they are very high, nearly 100% in combination, on smaller datasets as well. Note here that the scale on the y-axis is smaller than Figure 9a, and that the success rates are consistently above 80% for both rules.

In summary, the evaluation of the pruning rules shows that, indeed, they are responsible for pruning the majority of the recursive calls, which, in turn, was shown to reflect the query performance. As the size of C grows, so too do the pruning rule success frequencies. Consequently, PrioReA is able to outperform the competing algorithms consistently and scale gracefully.

7. CONCLUSION AND OUTLOOK

In this paper, we introduced the Sky-not query to empower a user to better understand skyline results. Given a point p and a constrained skyline query, the Sky-not query returns the minimal change to the user's constraints that places p in the skyline. This can be used both to understand how competitive p is relative to the skyline and to dynamically adapt queries to fit user needs and expectations.

Towards this, we first conducted a theoretical study of Sky-not queries, showing that the space of possible solutions can be discretized and the set of relevant input points can be dramatically reduced. We also showed Sky-not queries are dual to the minimum skyline distance problem, implying techniques for each can be shared. We then presented two novel algorithms: BRA transforms the Sky-not query one highly resembling a new skyline instance, and adapts the BNL skyline algorithm; PrioReA uses theoretically motivated pruning rules in a recursive framework. The latter we showed has excellent empirical performance, largely on account of the success rate of the pruning rules.

We focused in this paper on improving the usefulness of skyline queries that include selection, perhaps the most fundamental element of database queries. There is much interesting work to be done in expanding this research with other clauses, such as joins and aggregations, where there is yet more complexity for the user to understand query results. Furthermore, there is a broad range of applications for skyline queries, such as road networks and social network graphs, and these settings may provide unique, specific challenges for supporting user understanding of skyline results.

8. ACKNOWLEDGMENTS

This research was conducted as part of the WallViz project (`http://wallviz.dk/`), supported by the Danish Council for Strategic Research, grant 10-092316. We would like to thank Darius Šidlauskas, Michael Lind Mortensen, and Kenneth S Bøgh for discussions that inspired this research.

9. REFERENCES

[1] Bartolini, I., Ciaccia, P., and Patella, M. Efficient sort-based skyline evaluation. *TODS 33*, 4 (2008), 31:1–49.

[2] Bidoit, N., Herschel, M., and Tzompanaki, K. Query-based why-not provenance with NedExplain. In *Proc. EDBT* (2014), pp. 145–156.

[3] Börzsönyi, S., Kossman, D., and Stocker, K. The skyline operator. In *Proc. ICDE* (2001), pp. 421–430.

[4] Chapman, A., and Jagadish, H. V. Why not? In *Proc. SIGMOD* (2009), pp. 523–534.

[5] Cheema, M. A., Lin, X., Zhang, W., and Zhang, Y. A safe zone based approach for monitoring moving skyline queries. In *Proc. EDBT* (2013), pp. 275–286.

[6] Chester, S., Mortensen, M. L., and Assent, I. On the suitability of skyline queries for data exploration. In *Proc. ExploreDB* (2014), pp. 7:1–6.

[7] Chester, S., Šidlauskas, D., Assent, I., and Bøgh, K. S. Scalable parallelization of skyline computation for multi-core processors. In *Proc. ICDE* (2015).

[8] Chomicki, J., Godfrey, P., Gryz, J., and Liang, D. Skyline with presorting. In *Proc. ICDE* (2003), pp. 717–719.

[9] He, Z., and Lo, E. Answering why-not questions on top-k queries. In *Proc. ICDE* (2012), pp. 750–761.

[10] Herschel, M., and Hernández, M. A. Explaining missing answers to SPJUA queries. *PVLDB 3*, 1–2 (2010), 185–196.

[11] Huang, J., Chen, T., Doan, A., and Naughton, J. F. On the provenance of non-answers to queries over extracted data. *PVLDB 1*, 1 (2008), 736–747.

[12] Huang, J., Jiang, B., Pei, J., Chen, J., and Tang, Y. Skyline distance: a measure of multidimensional competence. *Knowl. Inf. Syst. 34*, 2 (2013), 373–396.

[13] Islam, M. S., Zhou, R., and Liu, C. On answering why-not questions in reverse skyline queries. In *Proc. ICDE* (2013), pp. 973–984.

[14] Kim, Y., You, G.-w., and Hwang, S.-w. Ranking strategies and threats: a cost-based pareto optimization approach. *Distributed and Parallel Databases 26*, 1 (2009), 127–150.

[15] Lee, J., and Hwang, S.-w. Scalable skyline computation using a balanced pivot selection technique. *Inf. Syst. 39* (2014), 1–21.

[16] Lee, J., You, G.-w., Hwang, S.-w., Selke, J., and Balke, W.-T. Interactive skyline queries. *Inf. Sci. 211*, 30 (2012), 18–35.

[17] Lu, H., and Jensen, C. S. Upgrading uncompetitive products economically. In *Proc. ICDE* (2012), pp. 977–988.

[18] Magnani, M., Assent, I., Hornbæk, K., Jakobsen, M. R., and Larsen, K. F. SkyView: a user evaluation of the skyline operator. In *Proc. CIKM* (2013), pp. 2249–2254.

[19] Meliou, A., Gatterbauer, W., and Moore, K. F. Why so? or Why no? Functional causality for explaining query answers. Tech. Rep. UW-CSE-09-12-01, University of Washington, Seattle, WA, USA, 2009.

[20] Papadias, D., Tao, Y., Fu, G., and Seeger, B. Progressive skyline computation in database systems. *PODS 30*, 1 (2005), 41–82.

[21] Peng, Y., and Wong, R. C.-W. Finding competitive price. In *Proc. SIGSPATIAL* (2013), pp. 144–153.

[22] Santoso, B. J., and Chiu, G.-M. Close dominance graph: an efficient framework for answering continuous top-k dominating queries. *TKDE PP* (2013), 1–14.

[23] Tan, K.-L., Eng, P.-K., and Ooi, B. C. Efficient progressive skyline computation. In *Proc. VLDB* (2001), pp. 301–310.

[24] Tran, Q. T., and Chan, C.-Y. How to ConQueR why-not questions. In *Proc. SIGMOD* (2010), pp. 15–26.

[25] Wan, Q., Wong, R. C.-W., Ilyas, I. F., Ozsu, M. T., and Peng, Y. Creating competitive products. *PVLDB 2*, 1 (2009), 898–909.

[26] Wu, P., Agrawal, D., Eğecioğlu, Ö., and El Abbadi, A. DeltaSky: optimal maintenance of skyline deletions without exclusive dominance region generation. In *Proc. ICDE* (2007), pp. 486–495.

[27] Zhang, S., Mamoulis, N., and Cheung, D. W. Scalable skyline computation using object-based space partitioning. In *Proc. SIGMOD* (2009), pp. 483–494.

[28] Zong, C., Yang, X., Wang, B., and Zhang, J. Minimizing explanations for missing answers to queries on databases. In *Proc. DASFAA* (2013), pp. 254–268.

Efficient Processing of Hamming-Distance-Based Similarity-Search Queries Over MapReduce [*]

Mingjie Tang †, Yongyang Yu †, Walid G. Aref †, Qutaibah M. Malluhi ‡, Mourad Ouzzani ⋆

† *Computer Science, Purdue University,* ‡ *Qatar University,* ⋆ *Qatar Computing Research Institute*

{tang49,yu163,aref}@cs.purdue.edu, qmalluhi@qu.edu.qa, mouzzani@qf.org.qa

ABSTRACT

Similarity search is crucial to many applications. Of particular interest are two flavors of the Hamming distance range query, namely, the Hamming select and the Hamming join (Hamming-select and Hamming-join, respectively). Hamming distance is widely used in approximate near neighbor search for high dimensional data, such as images and document collections. For example, using predefined similarity hash functions, high-dimensional data is mapped into one-dimensional binary codes that are, then linearly scanned to perform Hamming-distance comparisons. These distance comparisons on the binary codes are usually costly and, often involves excessive redundancies. This paper introduces a new index, termed the *HA-Index*, that speeds up distance comparisons and eliminates redundancies when performing the two flavors of Hamming distance range queries. An efficient search algorithm based on the *HA-index* is presented. A distributed version of the *HA-index* is introduced and algorithms for realizing Hamming distance-select and Hamming distance-join operations on a MapReduce platform are prototyped. Extensive experiments using real datasets demonstrates that the *HA-index* and the corresponding search algorithms achieve up to two orders of magnitude speedup over existing state-of-the-art approaches, while saving more than ten times in memory space.

1. INTRODUCTION

Hamming-distance search over big data plays an important role in a large variety of applications. For example, widely used search engines, such as Google, Baidu, and Bing, use Hamming-distance search in their image content-based search engines that usually index billions of images (e.g., refer to [1]). Typically, each image is modeled by a high-dimensional vector of extracted features, e.g., color histograms, texture features, and edge orientation. Then, based on the learned similarity hash function, e.g., as in [1, 2, 3], each image is converted into a binary code. Given a query image that gets modeled with the same high-dimensional vector of features, the search engine maps it into a binary code and performs a Hamming-distance search to find images whose binary codes have a Hamming distance smaller than a given threshold $\hat{h}$ from the query image. Hamming search is also widely used to detect duplicate web pages in applications, e.g., web mirroring, plagiarism, and spam detection [4]. A similarity hash function [5] is applied to map a high-dimensional vector into a binary code, then a Hamming-distance range search finds web documents that are similar to the query document.

Typically, computing the Hamming distance between two binary codes is performed by an Exclusive-Or operation (XOR, for short) that is followed by a count operation to sum up the number of ones in the XOR result. The number of ones corresponds to the number of differing bits between the two binary codes. Thus, a linear scan over the binary codes of the underlying dataset takes place to perform the XORing, the counting, and the ranking to retrieve the objects within a certain range of t_q (i.e., the ones within the predefined Hamming distance threshold $\hat{h}$). Due to the linear scan, this approach is slow. When joining two tables via a Hamming distance predicate, the linear scan approach induces a quadratic cost to evaluate the join. An efficient indexing of the binary codes is called for to perform the Hamming range query and avoid a complete scan over the underlying dataset, while remaining low on memory usage.

The Hamming distance problem [6, 7] is first studied for small distance thresholds, i.e., $\hat{h} = 1$. An algorithm proposed by Manku et al. [4] uses multiple hash tables to enhance query speed. However, duplicating the hash entries multiple times for the entire datasets is expensive and performance tends to degrade as a linear scan over tuples within a bucket is required. HEngine [8] extends Manku's algorithm to improve the query's speed with less memory. However, HEngine is sensitive to the Hamming distance threshold $\hat{h}$, and it needs to generate one-bit differing binary code with each query, then carry out several binary searches over sorted hash tables. Recently, MapReduce as a reliable distributed computing model has been adopted for handling a variety of similarity queries, e.g., [4, 9, 10, 11, 12]. Existing techniques for Hamming-distance queries cannot be easily extended for MapReduce. The reason is that most of the existing techniques use centralized multiple hash-table indexes. Because MapReduce needs to write intermediate data on disk when shuffling data between the mappers and the reducers, rearranging multiple indexes and multiple versions of the same data can be quite inefficient.

In this paper, we focus on two variants of the Hamming distance query problem, namely Hamming-distance-based select and Hamming-distance-based join (for short, Hamming-select and Hamming-join, respectively). We propose a new index, termed the HA-Index, that is designed to reduce redundant and duplicate dis-

[*]This work was supported by an NPRP grant 4-1534-1-247 from the Qatar National Research Fund and by the National Science Foundation Grants IIS 0916614, IIS 1117766, and IIS 0964639.

10.5441/002/edbt.2015.32

tance computations during the Hamming-distance search. The HA-Index assumes that the underlying datasets are preprocessed; data is mapped from the high-dimensional space into one-dimensional binary codes that are fixed-length strings of 0's and 1's. Then, the binary codes are sorted using the Gray ordering [13]. Sorting the binary codes in this way helps group together multiple binary codes that share a common substring or non-contiguous yet similar sequences of bits. By computing the distances between the query binary code and similar substrings, many redundant distance computations can be avoided.

The contributions of this paper are as follows.

- Based on properties of binary codes, we introduce two approaches to improve the performance of Hamming-select and Hamming-join. The first approach uses a simple Radix-tree index from the literature. The second approach is based on the HA-Index with both a static and a dynamic version. We also introduce the maintenance operations, i.e., build, insert, update, and search operations, for the dynamic HA-Index.
- For Hamming-joins over large and skewed data, we propose an efficient data partitioning technique for balancing data computations among servers, and introduce a distributed version of the HA-Index to reduce data shuffling inside MapReduce.
- We conduct an extensive experimental study using real datasets and demonstrate that the HA-Index (i) enhances the performance of Hamming-select and Hamming-join by two orders of magnitude over state-of-the-art techniques, and (ii) saves memory usage by more than one order of magnitude. We also evaluate how the proposed index improves approximate algorithms for kNN-select and kNN-join operations.

The rest of this paper proceeds as follows. Section 2 discusses related work. Section 3 presents the problem definition. Section 4 introduces the centralized-server approach for approximate Hamming-select and Hamming-join. Section 5 introduces the distributed version of the HA-Index using MapReduce and explains how Hamming-select and Hamming-join can be performed in MapReduce. Section 6 presents and discusses the experimental results. Finally, Section 7 contains concluding remarks.

2. RELATED WORK

Using the Hamming distance as a similarity metric has been studied in the theory community, e.g., [6, 7]. When the Hamming-distance query threshold is small, i.e., $\hat{h} = 1$, Yao et.al [7] propose an algorithm with $O(m \log\log(n))$ query time and $O(nm \log(m))$ space. Yao's algorithm recursively cuts the query binary code and each binary code in the dataset in half, and then finds exact matches in the dataset for the left or the right half of the query binary code. [14] demonstrates that similarity search in chemical information via the Tanimoto Similarity metric can be transformed into a Hamming-distance query.

Hamming-distance queries are attracting more attention for processing large volumes of data. A relatively recent work [4] uses multiple hash tables, and hence more space, to reduce the linear computation of the Hamming distance during query time. The idea behind this approach is that if two binary codes are within a Hamming distance $\hat{h}$, then at least one of the $\hat{h}$+1 segments are exact matches for two binary codes. This algorithm needs to replicate

Table 1: Symbols and their definitions

Symbol	Definition
$\mathbb{R}^d$	d-dimensional vector space
n, $\|R\|$	Number of tuples in dataset R
m, $\|S\|$	Number of tuples in dataset S
t_q	Query tuple
k	The required number of nearest neighbors
$\|\|t_i, t_j\|\|_h$	Hamming distance between tuples t_i and t_j
H	Similarity Hash function
U_i	Binary code for tuple t_i
$L = \|U\|$	Length of the binary code U
l_i	The ith bit in the binary code
$\hat{h}$	Hamming distance threshold
$\hat{h}$-select(t_q, S)	Hamming distance select for tuple t_q and datasets S
$\hat{h}$-join(R, S)	Hamming distance join between datasets R and S
N	Number of data partitions

the database multiple times, and it sorts each copy based on parts of the segments. The Hamming-distance computation is still performed in a linear fashion over tuples of the same bucket in a certain hash table. Thus, it fails to scale as data size increases. For performing a Hamming-join of two datasets, say R and S, [4] extends the sequential approach to MapReduce by broadcasting Table R into each server, then applying a sequential algorithm between R and S. This approach is subject to a very heavy shuffling cost and servers cannot work in a load-balanced way when data is skewed. HEngine [8] adopts a similar idea to that in [4], but uses approximate matching instead by generating multiple one-bit difference binary codes. The HEngine uses less memory but achieves limited performance speedup. HmSearch [14] is an exact matching approach that index over signature of the binary codes. The size of the index increases dramatically, because HmSearch need to generated large amount of unique signatures. If used in the context of MapReduce, the shuffling cost between the mappers and the reducers is expected to be expensive. Our proposed HA-Index extracts and groups similar binary codes from among the various tuples to reduce the cost of shuffling and hence is applicable to MapReduce as we illustrate later in this paper. Through data sampling, we partition that data in a way that uniformly distributes the dataset among the reducer servers and hence enables better load balancing. Experimental comparison with [4, 8] shows that our proposed HA-Index is two orders of magnitude faster and uses ten times less memory as illustrated in the experimental section of this paper.

Two related and popular operations to Hamming distance queries are the k-nearest-neighbor select (kNN-select) and k-nearest-neighbor join (kNN-join) [15, 16]. Given a dataset, say S, and a query focal point, say t_q, kNN-select finds in S the k-nearest-neighbors to t_q. Given two datasets, say R and S, R kNN-join S finds the k-nearest-neighbors in S for each tuple in R. In high-dimensional spaces and because of the curse of dimensionality [17], data-independent hash-based approximate kNN (e.g., locality sensitive hashing (LSH) [18]) has attracted attention as it can speed-up query execution while having acceptable error margins. Recently, data-dependent hashing has been proposed to learn the hash function, say $H()$, given the underlying dataset, e.g., as in [2]. There has been a plethora of work in learning good and representative hash functions, e.g., [2, 3, 1]. Given the learned similarity hash function $H()$, a tuple, say t_i, is mapped into its binary code, say U_i, i.e., $H(t_i) = U_i$. Afterwards, all the binary codes of the dataset R are scanned to find data tuples that are different from the query's binary code U_i by at most $\hat{h}$ bit-positions. If the answer set size is more than k, then only the k-closest answers are retained. However, if the size of the result set is less than k, then a larger distance threshold is estimated and the near neighbor query is repeated. The process is stopped when k or more answers are reported. Notice that the core of the method for approximate kNN search is a Hamming-distance query with a threshold $\hat{h}$. In our experiments, we use the state-of-the-art approach [2] to learn the hash function, and show how our proposed approach can speed up approximate kNN-select and kNN-join.

3. PRELIMINARIES

3.1 Hamming-distance-based Similarity Operations

We assume that data tuples represent points in a d-dimensional metric space, say $\mathbb{R}^d$. Given two data tuples, say t_i and t_j, let $||t_i, t_j||$ be the distance between t_i and t_j in $\mathbb{R}^d$. The Hamming distance between t_i and t_j, denoted by $||t_i, t_j||_h$, helps in retrieving the tuples in a dataset that are within some threshold from an input tuple, either t_i or t_j in this case. Table 1 summarizes the symbols used in this paper.

DEFINITION 1. *Hamming-distance-based Similarity Select [4] (referred to as Hamming-select, for short): Given a query tuple, say t_q, and a dataset, say S, with its corresponding collection of binary codes, denoted by U_S, and an integer, say $\hat{h}$, that represents the similarity threshold for the Hamming distance, Hamming-select identifies a subset from S, denoted by $\hat{h}$-select(t_q, S) for short, where $\forall o \in \hat{h}$-select(t_q, S), $||o, t_q||_h \leq \hat{h}$.*

Similarly, we define the Hamming-distance-based similarity join as follows.

DEFINITION 2. *Hamming-distance-based Similarity Join (referred to as Hamming-join, for short): Given two collections of binary codes, say U_R and U_S, that correspond to two datasets, say R and S, respectively, and an integer, say $\hat{h}$, that represents the similarity threshold for the Hamming distance, Hamming-join identifies the set $\hat{h}$-join(R, S) of tuple pairs such that $(t_i, t_j) \in \hat{h}$-join(R, S) iff $t_i \in R$ and $t_j \in S$ and $||t_i, t_j||_h \leq \hat{h}$.*[1]

EXAMPLE 1. *Consider the set of binary codes given in Table 2a and a Hamming distance threshold $\hat{h} = 3$. The query tuple t_q has a binary code "101100010". The output of the Hamming-distance-based similarity select is $\{t_0, t_3, t_4, t_6\}$. Using the same Hamming distance threshold $\hat{h}$, the output of the Hamming-distance-based similarity join for the datasets in Tables 2b and 2a is $\{(r_0, t_0), (r_0, t_3), (r_0, t_4), (r_0, t_6)\}$,$\{(r_1, t_0), (r_1, t_3), (r_1, t_4), (r_1, t_6)\}$, $\{(r_2, t_3)\}$.*

From the example above, one can produce the output set by simply scanning the table one tuple at a time, performing Hamming distance calculation via the XOR operation, and reporting the tuple as an output if the computed Hamming distance is smaller than or equal to $\hat{h}$. If $|S| = n$, then the cost of computing Hamming-select consists of $O(n)$ tuple reads and $O(n)$ Hamming-distance computations. Similarly, the cost of computing Hamming-join between the two datasets R and S, where $|R| = m$ and $|S| = n$ respectively, with a nested-loop join algorithm, consists of $O(mn)$ tuple reads and $O(mn)$ Hamming-distance computations. The focus of this paper is to develop a Hamming-distance-based tree index to reduce the above costs.

[1]Different from the kNN-join, $\hat{h}$-join for datasets R and S is symmetric, i.e., $\hat{h}$-join(R, S)= $\hat{h}$-join(S, R).

Table 2: An example illustrating a Hamming-distance query.

(a) Table S

tuple	**binary** U
t_0	001 001 010
t_1	001 011 101
t_2	011 001 100
t_3	101 001 010
t_4	101 110 110
t_5	101 011 101
t_6	101 101 010
t_7	111 001 100

(b) Table R

tuple	**binary** U
r_0	101 100 010
r_1	101 010 010
r_2	110 000 010

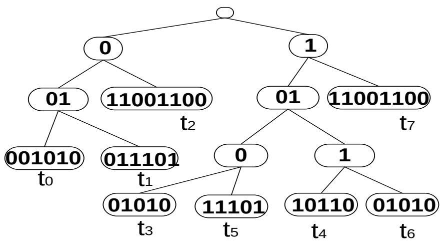


Figure 1: Radix Tree

4. HAMMING-SELECT ALGORITHMS

In this section, we first introduce the basic concept and principles of binary hash codes, and illustrate the Radix-Tree-based approach. We then introduce two variants of our proposed HA-Index, namely the static and dynamic HA-indexes along with their associated algorithms.

4.1 Properties of Binary Codes

DEFINITION 3. *A binary code $\hat{U}$ is said to be a fixed-length substring ($FLSS$) of another binary code U if $|U| = |\hat{U}|$ and there exist i and j, $1 \leq i$, $i + j \leq |U|$ such that $\forall i, i \leq v \leq i + j$, and $U[v] = \hat{U}[v]$. Thus, only the bits between i and $i + j$ are the same and all the remaining can be any combination of* 0*s and* 1*s.*

For example, consider Tuple t_0 in Table 2a. Let $\cdot$ denote a 0 or a 1. Based on the above definition, $\hat{U}$="$\cdot\cdot\cdot\cdot$**0101** $\cdot$ " is one $FLSS$ of t_0's binary code "0011**0101**0". Alternatively, $\hat{V}$ ="101 $\cdot\cdot\cdot\cdot\cdot\cdot$" is not an $FLSS$ of t_0's binary code.

DEFINITION 4. *A binary code, $\hat{U}$, is the fixed-length SubSequence ($FLSSeq$, for short) of a binary code U if there exists a strictly increasing sequence of indices of U such that $\forall j \in \{1, 2, \ldots, k'\}$, we have $U[j] = \hat{U}[j]$ and $|U| = |\hat{U}|$.*

For example, $\hat{U}$="$\cdot\cdot\cdot$ **0** $\cdot$ **1** $\cdot$ **1** $\cdot$ " is one possible $FLSSeq$ of t_0's binary code "001**0**0**1**0**1**0" in Table 2a. Thus, $\hat{U}$ belongs to Set $FLSSeq$ of Tuple t_0. To compute the Hamming distance between an $FLSSeq$ and a query binary code, we only count the bit difference in the corresponding effective bit positions. For instance, if one $FLSSeq$ is $\hat{U}$="$\cdot\cdot\cdot$ **0** $\cdot$ **1** $\cdot$ **1** $\cdot$ " and the query binary code is U="001**0**0**1**0**1**0", the Hamming distance $||\hat{U}, U||_h$=2.

PROPOSITION 1. ***Hamming Downward Closure Property*** *A binary code $U \in \hat{h}$-select(t_q, S) iff each $FLSS$ of U, say U_{FLSS}, (each $FLSSeq$ of U, say U_{FLSSeq}, respectively) meets the condition $||t_q, U_{FLSS}||_h \le \hat{h}$ ($||t_q, U_{FLSSeq}||_h \le \hat{h}$, respectively).*

We omit the proof for simplicity. Instead, we illustrate the above proposition using the following example.

EXAMPLE 2. *Refer to the Hamming-distance query in Example 1 and Table 2. Suppose that the Hamming-distance threshold $\hat{h} = 2$. Consider the following example cases:*

- *Case 1: Given a query binary code t_q = "110010010", since one FLSS, U_{FLSS} = "$001 \cdots\cdots$", is the binary code of an $FLSS$ for both t_0 and t_1 and $||U_{FLSS}, t_q||_h \ge 3$, then neither t_0 nor t_1 can belong to $\hat{h}$-select(t_q, S).*
- *Case 2: Given a query binary code t_q = "110110010", the binary code "$\cdot$ 11001100" is an FLSS (U_{FLSS}) for both t_2 and t_7, $||U_{FLSS}, t_q||_h \ge 3$, thus, neither t_2 nor t_7 can belong to $\hat{h}$-select(t_q, S).*
- *Case 3: Given a query binary code t_q = "110100010", the binary code "$1010 \cdot 1 \cdots$" is an $FLSSeq$ for both t_3 and t_5, $||U_{FLSSeq}, t_q||_h \ge 3$, therefore, neither t_3 nor t_5 can belong to $\hat{h}$-select(t_q, S).*

4.2 Radix-Tree-Based Approach

The idea behind using a Radix-Tree index (also termed the PATRICIA trie) [19] is to merge the XOR operations for various binary codes if they happen to share $FLSS$s, e.g., similar to Case 1 of the example above. One XOR operation on a common $FLSS$ can be used to verify all participant tuples in this $FLSS$. Thus, we can build a prefix tree out of the binary codes. Based on the above closure property (Proposition 1), we can compute the Hamming distance with prefixes of the Radix-Tree from the root to find qualifying binary codes in a top-down fashion.

EXAMPLE 3. *Figure 1 gives the corresponding Radix-Tree for the binary codes in Table 2. From the Radix-Tree, Tuples t_0 and t_1 in Table 2 share the same FLSS U_{FLSS} = "$001 \cdots\cdots$". Given the query binary code t_q = "110010110" and a Hamming-distance query threshold $\hat{h} = 2$, both Tuples t_0 and t_1 can be discarded without computing the whole Hamming distance for all binary positions, because the Hamming distance from U_{FLSS} with the first three bits of t_q is bigger than the predefined threshold $\hat{h}$. Thus, processing the Hamming-distance-based select can stop early at the upper level of the Radix-Tree.*

Notice that although useful in the above example, the Radix-Tree-based approach has several disadvantages, mainly due to its prefix-sensitiveness. For example, Tuples t_2 and t_7 in Figure 1 are split into two branches in the Radix-Tree, although only the first bit in the two tuples is different while all their remaining bits are the same. Thus, the search path would go to different branches of the tree and redundant computations in these two branches cannot be avoided. In the worst case, if the binary codes in the Radix-Tree do not share common prefixes, then searching from the root will bring the computation cost as bad as $O(2^L)$, because it would go through every branch of the Radix-Tree. As a result, we propose the HA-Index to address the prefix-sensitivity of the Radix-Tree-based approach.

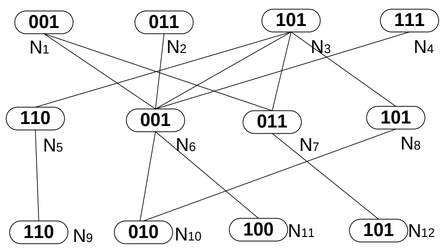


Figure 2: Static HA-Index

4.3 Static HA-Index

The idea behind the Static HA-Index is to share the common substrings, i.e., the maximal $FLSS$s, in contrast to sharing the common prefixes for the binary codes of the underlying dataset. Thus, redundant Hamming distance computations can be avoided. Recall Case 2 of Example 2, the $FLSS$ for t_2 and t_7 is "$\cdot$01101010". For the Radix-Tree-based approach in Figure 1, searching for the qualifying tuples would proceed to different paths, which introduces redundant computations. Thus, if we are able to realize an index that shares the common $FLSS$s, we would be able to avoid redundant and unnecessary Hamming-distance computations.

Static bit segmentation: We segment the binary codes into fixed-length contiguous substrings (called fixed-length segments). For instance, assuming that each segment is of Size 3, the binary code for tuple t_2 is divided into three segments, "011", "001" and "100". The path along these segments can be traced via an undirected path. For example, the path that corresponds to tuple t_2 is illustrated in Figure 2 where it connects Nodes N_2 to N_{11} via Intermediate Node N_6. Meanwhile, the path of Tuple t_7 includes Nodes N_4, N_6 and N_{11}. Thus, Tuples t_2 and t_7 can share the same vertex nodes N_6 and N_{11}. While traversing the index, the Hamming-distance computation for Nodes N_6 and N_{11} will be performed only once. In the next section, we demonstrate how the Static HA-Index can be used to evaluate both the Hamming-select and Hamming-join operations.

The static HA-Index has several limitations though. Both the height and the length of the paths in the Static HA-index are sensitive to the segment size. Because the segment sizes are fixed, it is possible to miss common bit substrings that do not align to segment boundaries. Also, both the Radix-Tree and the static HA-Index optimize for the $FLSS$s of the binary codes. An index that would support $FLSSeq$s, in contrast to just the $FLSS$s (recall that the $FLSS$s are subsets of the $FLSSeq$s), would allow for more shared distance computations and hence additional savings. Consider Case 3 of Example 2. Both the Radix-Tree and Static-HA-Index approaches fail to capture the common $FLSSeq$ between t_3 and t_5. In the next section, we introduce the Dynamic HA-Index to address these limitations.

4.4 Dynamic HA-Index

DEFINITION 5. *Gray Order: is an ordering of the binary codes such that consecutive binary codes differ only by one bit, i.e., the Hamming distance between two consecutive binary codes that are sorted according to the Gray order is equal one [13].*

PROPOSITION 2. ***Gray Order and Clustering***: *When the binary codes are ordered based on the Gray order, data tuples are naturally clustered [20],i.e., the Hamming distance between consecutive ordered binary codes is small as the consecutively ordered binary codes share common $FLSSeq$s.*

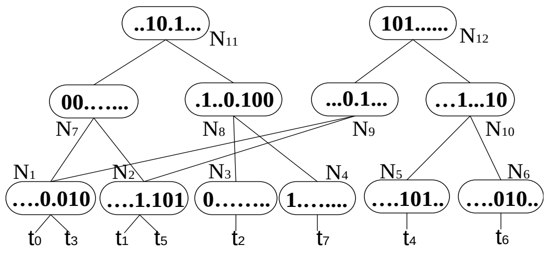


Figure 3: Dynamic HA-Index

For instance, the data tuples in Table 2 can be ordered based on the Gray order of their corresponding binary codes in descending order, and the resulting sorted order is $\{t_0, t_1, t_2, t_7, t_4, t_6, t_3, t_5\}$. Observe that the sorted binary codes provide two important properties, namely the downward closure and the clustering properties, that facilitate efficient Hamming-distance-based query processing. Thus, our aim is to realize an index structure that preserves and leverages these properties. The Dynamic HA-Index will strategically divide the binary codes into segments (i.e., sequences of data points that are close in their binary values according to the Gray order). As such, the clustering property is preserved to ensure that nodes with similar $FLSSeq$s are close to each other in the index. For example, Tuples t_2 and t_7 are ordered next to each other, and these properties can overcome the prefix-sensitivity of the Radix-Tree-based approach.

In the Dynamic HA-Index, the leaf nodes store data tuples while the non-leaf nodes store the $FLSSeq$s of the children nodes. Refer to Figure 3 for an illustration. Internal node N_1 represents the $FLSSeq =$ "$\cdots 0{\cdot}010$" of Tuples t_0 and t_3. Internal node N_2 represents the $FLSSeq =$ "$\cdots 1{\cdot}101$" that is common to both Tuples t_1 and t_5. Furthermore, Internal Node N_7 represents the $FLSSeq$ for Nodes N_1 and N_2. Notice, all the descendants of an HA-Index node can be safely discarded from further Hamming-distance computations if the node's corresponding $FLSSeq$ does not qualify the Hamming-distance threshold, thereby reducing computation overheads.

4.5 Dynamic HA-Index Manipulation

The primary objective of all the Dynamic HA-Index manipulation algorithms, including build, delete, and insert, is to maintain the $FLSSeq$ properties of the index while keeping the size of the index reasonably small.

Bulkloading builds the Dynamic HA-Index in a bottom-up fashion. It has two steps. The first step sorts all the data tuples according to the Gray order of their nondecreasing binary codes. The second step scans these tuples sequentially using a sliding window with w slots to form index nodes. Algorithm 1 illustrates the pseudo-code to build the Dynamic HA-Index. A queue is initialized to store the temporary nodes from the window (Line 2). From the tuples within a window, Function `extractFLSSeq` extracts the maximal $FLSSeq$s from the tuples' binary codes to form new parent nodes (Line 5), and denotes the new binary code of the child node. Then, the new temporal node is inserted into the queue (Line 7). For instance, Tuples t_0 and t_1 share the same $FLSSeq =$ "$0010 \cdot 1 \cdots$". Thus, this $FLSSeq$'s corresponding new node is formed and is inserted into the queue. To save memory storage, Function `extractFLSSeq` captures the binary code of t_0 as "$\cdots \cdot 0 \cdot 010$". Therefore, the non-leaf nodes with the same $FLSSeq$ are consolidated into one node. Hence, Tuple t_3 would be denoted with the same binary code as that of t_0, and would share

Algorithm 1: H-Build

```
   Input: $T$: Set of data points, $w$: Window, $md$: Depth of HA-index, $s$:
          Sliding window size
   Output: $HA$:HA-Index for dataset $T$
1  Sort $T$ based on the non-decreasing Gray order of the tuples' binary
   codes;
2  q: Queue;
3  for each data element $t_i$ of $T$ inside Window $w$ do
4      var $n$, $\hat{n}$: Node;
5      $n$, $\hat{n} \leftarrow$ extractFLSSeq($t_i, \cdots, t_{i+w}$); // $n$, the parent node of $\hat{n}$
6      if $\hat{n}$ is new then
7          insert $\hat{n}$ into the current level of the HA-Index.
8      end
9      else
10         update $\hat{n}$'s frequency
11     end
12     if $n$ is not empty then
13         q.enqueue($n$);
14     end
15     else
16         put Tuple $t_i$ inside Window $w$ into the top level of the
           HA-Index;
17     end
18     w← w+s; //sliding the window
19 end
20 var $d$:0, $begin$:0, $end$:q.size;
21 while $q$ is not empty and $d \leq md$ do
22     // Process similar to Lines 4-18
23     // Use two pointers for q to record the HA-Index depth d
24 end
```

the same binary codes. Notice that we record the frequency of each node (Line 6-11). For example, Node N_1 represents the binary code for t_0 and t_3. Thus, the frequency for N_1 is 2. If tuples inside the window do not share any $FLSSeq$ among each other, these tuples are linked to the top level of the HA-Index (Line 16). The window continues to slide until all the data points are scanned in the first round. Lines 21-24 merge the internal nodes as Lines 4-18 and we can use two pointers `begin` and `end` for the queue to indicate the depth. The building process continues until the desired depth is reached.

In addition, more than one leaf node can be linked to the same internal node, e.g., Tuples t_1 and t_5 are linked to Internal Node N_1 in Figure 3. Thus, we build a hash table for the bottom node, e.g., N_1, where the key is the leaf node's binary codes, and value is the tuple's ID. Naturally, if users only want to learn the qualifying binary codes, then there is no need to keep the leaf nodes of The HA-Index. An HA-Index without leaf nodes could save the overhead of building hash tables, and can be used in MapReduce Hamming-join as in Section 5.

Deletion removes a tuple with its corresponding binary code from a Dynamic HA-Index. Algorithm 2 gives the corresponding process. First, a leaf node that contains the tuple to be deleted is located by depth-first search using the tuple's binary code as the search key. One stack is used to denote the unexplored paths. Function `bitmatch` tests whether one binary code is the $FLSS$ or $FLSSeq$ of the deleted tuple (Lines 3 and 14). Then, the tuple is removed from the HA-Index. After deletion, the frequency of the corresponding node needs to be decremented (Lines 5 and 16). If one node contains 0 or less entries, it is removed.

Inserting a new data tuple into a Dynamic HA-Index is similar to the deletion process. Insertion uses a depth-first search to locate the corresponding leaf node, then the search process looks for the leaf node that shares the maximal $FLSSeq$ with the newly inserted data tuple. If no such leaf node is found, we put the newly inserted data

Algorithm 2: H-Delete

```
Input: $t_q$: Deleted query tuple, HA: HA-Index for queried dataset
s: Stack;
for each top level node $n_i$ in HA do
    if bitmatch($t_q$, $n_i$) then
        $s$.push($n_i$);
        $n_i$.frequency ← $n_i$.frequency-1 ;
        remove $n_i$ from HA if $n_i$.frequency is 0 ;
    end
end
while s is not empty do
    var n: Node;
    s.pop(n);
    if n is a non-leaf node then
        for all child nodes c of n do
            if bitmatch($t_q$, $n_i$) then
                $s$.push($n_i$);
                $n_i$.frequency ← $n_i$.frequency-1 ;
                remove $n_i$ from HA if $n_i$.frequency is 0 ;
            end
        end
    end
    else
        break;
    end
end
```

tuple into a temporary buffer. When the buffer reaches a predefined maximum size, a process similar to H-Build is invoked to append these newly inserted tuples into the existing HA-Index. We omit these details here for brevity as they are similar to Algorithms H-Delete and H-Build.

4.6 HA-Index Query Processing

With the dataset organized in an HA-Index, H-Search traverses the index to visit the relevant index nodes in a breadth-first order with a queue to keep track of the unexplored qualifying paths that match the query's binary code. Algorithm 3 gives the pseudocode for H-Search. Initially, H-Search fetches the index nodes/data points from the top level of the HA-Index (Lines 2-6). If the Hamming distance between the query tuple and the pattern of the corresponding node is smaller than the threshold $\hat{h}$, then the node is inserted into the queue. For the non-top level nodes, in each round, the binary code of a node is examined against the query binary code by invoking a Hamming-distance computation. If its corresponding Hamming distance is smaller than the threshold (Line 12), the node is further explored (Lines 13-17). When a leaf node of the HA-Index is reached, the qualified data tuples are collected and are inserted into ret (Line 23-25). The algorithm terminates when all the entries from the qualifying nodes are examined.

To illustrate the H-Search Algorithm, consider the tuples in Table 2a. Figure 3 gives the corresponding HA-Index. The execution trace is given in Table 3. Suppose that the query binary code is t_q = "010001011" and the Hamming-distance threshold is 3. Initially, the Hamming distance between t_q and the top-level entries, i.e., $||N_{11}, t_q||_h = 1$ and $||N_{12}, t_q||_h = 3$, where both are no bigger than 3. Thus, Nodes N_{11} and N_{12} are pushed into the queue, and ret is still empty. Next, the children nodes of N_{11}, i.e., Nodes N_7 and N_8 are visited. The Hamming distances $||N_7, t_q||_h = 1$ and $||N_8, t_q||_h = 4$ are computed. As a result, the corresponding qualifying binary codes for Nodes N_{11} and N_7 are combined, which results in the pattern "$0010 \cdot 1 \cdots$". Thus, $[N_7, N_{11}]$ is put into the queue. But Node N_8 is discarded from the qualifying candidates because the path $N_{11} \to N_8$ has a combined Hamming distance $||N_{11}, t_q||_h + ||N_8, t_q||_h > 3$. Then, N_{12} is explored and its children nodes(e.g.,N_9 and N_{10}) are visited. According to the Hamming-distance closure properties, $[N_9, N_{12}]$ is inserted into the queue as well, while N_{10} is discarded. The H-Search process continues until the queue is empty as shown in Table 3. Finally, Tuple t_0 is reported as one output tuple qualifying the query. Notice that each node maintains a *visited* flag to indicate whether the node has already been visited or not. This helps avoid redundant Hamming-distance computations. For example, Nodes N_1 and N_2 are already visited. Therefore, we do not need to compute the Hamming distance for both nodes again, and hence avoid unnecessary distance computation overhead. In addition, Algorithm H-Search for the dynamic HA-Index can be applied to the static HA-Index, and thus is not repeated in the paper.

Algorithm 3: H-Search

```
Input: $t_q$: Query tuple, $\hat{h}$: Hamming distance query threshold, HA:
       HA-Index for queried dataset
Output: ret: Qualified tuple in HA within Hamming distance $\hat{h}$ from
        tuple $t_q$
q: Queue.
for each top level node $n_i$ in HA do
    if hdist($t_q$, $n_i$) ≤ $\hat{h}$ then
        $n_i$.h ← hdis($t_q$, c);
        $q$.enqueue($n_i$);
    end
end
while q is not empty do
    var n:Node;
    q.dequeue(n);
    if n is a non-leaf node then
        for all children node c of n do
            if (hdis($t_q$, c)+n.h)≤ $\hat{h}$ then
                var m:Node;
                m.b ← combine(c.b, n.b); //combine binary code of
                  c and n
                m.h ← hdis($t_q$, c)+n.h; //update Hamming distance
                m.children ← c.children ;
                $q$.enqueue($m$);
            end
        end
    end
    else
        var binary ← getBinary(n);
        var tuple ← gettuple(binary);
        ret.insert(tuple);
    end
end
output ret;
```

Table 3: Sample execution trace for H-Search that corresponds to searching the dataset in Table 2a given the query binary code t_q = "010001011"

Queue	**Qualified tuples** ret
N_{11}, N_{12}	$\emptyset$
$N_{12}, [N_7, N_{11}]$	$\emptyset$
$[N_7, N_{11}], [N_9, N_{12}]$	$\emptyset$
$[N_9, N_{12}]$	t_0
$\emptyset$	t_0

4.7 Analysis

Example 4. *Assume that we have eight tuples t_0 = "000", t_1 = "001", t_2 = "010", $\cdots$, and t_7 = "111", where all binary codes are distinct. At most 3 bits are needed to represent all the tuples, i.e., the length L of the hash values is 3. According to the H-Build process with Window Size of 2, the output HA-Index is illustrated in Figure 4.*

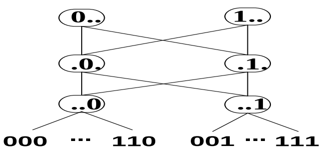


Figure 4: Full binary codes and the corresponding HA-Index

Observe that the number of internal nodes of this HA-index is 6, and the number of edges is 8. Based on the breadth-first-search strategy of the H-Search algorithm, the worst search cost is bounded by the number of internal nodes and the number of edges, denoted by $|V|$ and $|E|$, respectively. Refer to Figure 4 for illustration. The search cost is at worst 14. Suppose that the number of distinct binary codes is n_d, and $n_d = 2^L$. An HA-Index for this example is illustrated in Figure 4. The reason is that the $FLSSeq$ for the binary codes in the same window is maximized with Length $L-1$, and this $FLSSeq$ also shares the maximum similar patterns with its neighboring $FLSSeq$. Therefore, for the dataset with $n_d = 2^L$ data points and the built HA-Index as in Figure 4, the number of internal nodes $|V| = 2L$ or $|V| = 2\log_2 n_d$, and number of edges $|E| = 4(L-1)$ or $|E| = 4(\log_2 n_d - 1)$. This can be proven via induction (Details are omitted for brevity). Thus, the worst case for H-Search on this HA-Index is $|V| + |E| = 2\log_2 n_d + 4(\log_2 n_d - 1)$, i.e., is $O(\log_2 n_d)$. This indicates that H-Search can achieve the best performance under this scenario. We will discuss more general cases later.

Window size We discuss the relationship between window size, say as w, and binary string length L. Inspired by the previous extreme example, it is desirable that the n tuples can span the space of binary strings of L bits. L can be chosen such that $L = \lceil \log_2 n \rceil$, i.e., $2^{L-1} < n \le 2^L$. Thus, if n is closer to 2^L, then the corresponding HA-Index is closer to the extreme case in our motivating example above. On the other hand, the smallest value for n is $2^{L-1} + 1$, and this is the worst case, i.e., the sparsest distribution of tuples on the space of binary strings of Length L. For the simplicity of discussion, we assume that the hashed binary strings are uniformly distributed.

Under the above assumption, the maximum Hamming distance L_m for a window of size of w satisfies $\lceil \log_2 w \rceil \le L_m \le L$. If $L_m = L$, then the binary strings in the same window cannot be merged together since no shared bit position exists. Therefore, a careful choice should be made on the window size w. The extreme case when $w = n$ is apparently a bad choice since no sharing pattern can be extracted from the window. A similar argument applies for $w = 1$. For smaller values of w, many internal nodes are generated and this results in indexes with larger heights. A suggested value for the window size w is $w = 2^{\lceil L/2 \rceil}$ when $n \approx 2^L$. Suppose that $w = 2^{\lceil L/2 \rceil}$, then the maximum Hamming distance L_m in each window satisfies $\lceil L/2 \rceil \le L_m \le L$.

Number of nodes in an HA-Index If $n \approx 2^L$, suppose that there are only few windows with $L_m = L$ and we denote the number of these binary codes within that window as δ_1. Since the leaves share about half of the bits in their binary codes, this results in a number of $2^{\lceil L/2 \rceil} + \delta_1$ of internals nodes 1-level higher above the leaves, where $\delta_1 \ll 2^{\lceil L/2 \rceil}$. With the HA-index progressively growing, a higher level with $2^{\lceil L/4 \rceil} + \delta_2$ internal nodes can be built where $\delta_2 \ll \delta_1$. In the same way, the HA-index grows to the highest level with $2^{\lceil L/2^h \rceil} + \delta_h$ uppermost internal nodes, where h is the height of the index. Thus, the total number of nodes $|V|$ in the HA-index can be estimated by:

$$\begin{aligned}|V| &= 2^{\lceil L/2 \rceil} + 2^{\lceil L/4 \rceil} + \cdots + 2^{\lceil L/2^h \rceil} + \sum_{i=1}^{h} \delta_i \\ &= 2^{\lceil \log_2 n^{1/2} \rceil} + 2^{\lceil \log_2 n^{1/4} \rceil} + \cdots + \sum_{i=1}^{h} \delta_i \\ &< 2 \times 2^{\lceil \log_2 n^{1/2} \rceil} + \sum_{i=1}^{h} \delta_i \\ &< 2 \times 2^{\lceil \log_2 n^{1/2} \rceil} \\ &= O(\sqrt{n}).\end{aligned}$$

We can safely ignore the delta part since the summation is negligible compared to the dominant term.

If $n \approx 2^{L-1}$, then the window size w needs to shrink to a proper length. Based on the assumption of uniform distribution of the binary strings and Gray ordering, a proper window size can be set to $w = 2^{\lceil L/4 \rceil}$. The maximum Hamming distance L_m within a window satisfies $\lceil L/4 \rceil \le L_m \le L$. A similar analysis suggests that the number of internal nodes $|V'|$ satisfies:

$$\begin{aligned}|V'| &= 2^{\lceil L/4 \rceil} + 2^{\lceil L/4^2 \rceil} + \cdots + 2^{\lceil L/4^h \rceil} + \sum_{i=1}^{h} \delta'_i \\ &= 2^{\lceil \log_2 n^{1/4} \rceil} + 2^{\lceil \log_2 n^{1/4^2} \rceil} + \cdots + 2^{\lceil \log_2 n^{1/4^h} \rceil} + \sum_{i=1}^{h} \delta'_i \\ &= O(\sqrt[4]{n}).\end{aligned}$$

Number of Edges in an HA-Index For the number of edges in an HA-index, there are two extreme cases. Suppose that $n \approx 2^L$ and we have already discussed that the two levels above the leaves contain $2^{\lceil L/2 \rceil}$ and $2^{\lceil L/4 \rceil}$ internal nodes, respectively. The worst case is that each of the $2^{\lceil L/2 \rceil}$ nodes connects to each of the $2^{\lceil L/4 \rceil}$ nodes. This induces about $2^{3L/4}$ edges. Similarly, the edge number can be estimated,

$$|E| = 2^{3L/4} + 2^{3L/8} + \cdots + 2^{3L/2^{h+1}} < 2 \times 2^{3L/4} = O(\sqrt[4]{n^3}).$$

On the other hand, the best estimate is that there are no cross edges between the children and different parents. For this case, a lower bound of the number of edges is $O(\sqrt{n})$, which is similar to the number of vertices.

Query Cost and Storage Space of the HA-Index The cost of H-Search is bounded by the number of nodes and edges, i.e., $|V| + |E|$. Therefore, the worst cost for H-Search is traversing all the edges and nodes in the HA-index. This indicates that H-Search can be bounded in the range $[O(\sqrt{n}), O(\sqrt[4]{n^3})]$. Meanwhile, besides the storage of the leaf nodes, the space usage of the HA-index also depends on the sum of the number of nodes and edges, i.e., $[O(\sqrt{n}), O(\sqrt[4]{n^3})]$. Compared to the state-of-the-art approaches [4, 8], the HA-Index does not need to maintain several copies of the dataset. Thus, it can be kept in memory for fast query processing. Furthermore, the internal nodes of the HA-Index store enough binary information for the whole dataset, and hence introduce low overhead to broadcast an HA-Index to each server.

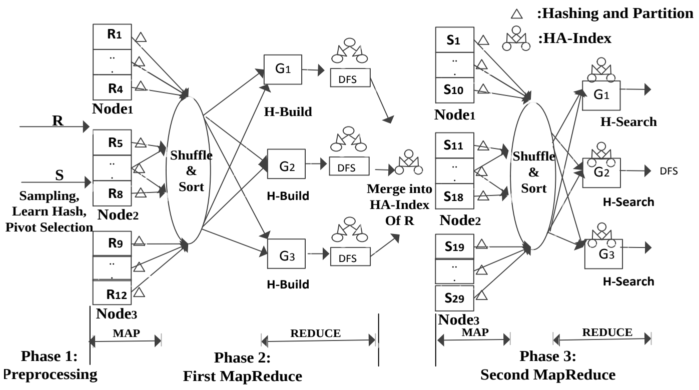


Figure 5: An overview of Hamming-join processing in MapReduce.

5. PARALLEL ALGORITHM FOR HAMMING-JOIN

To process Hamming-join on two datasets, say R and S, one straightforward approach is to build an HA-Index for R, then execute H-Search on the built index for each tuple of S. However, to build an HA-Index for R, sorting R would be slower as R gets larger. Secondly, executing H-Search between each tuple of S and the HA-Index for R would make the query time bounded by the number tuples in S. In this section, we address these limitations of the centralized environment and introduce Hamming-join on the MapReduce platform [21].

To support Hamming-join over MapReduce, we focus on two important issues. First, load balancing is important because the slowest mapper or reducer determines the job running time. Secondly, data shuffle from the mappers to reducers usually results in large disk I/O and network communication costs that heavily influences the run-time performance. Therefore, we not only need to reduce the data shuffle cost, but also make sure data partitions in each mapper or reducer are well balanced.

5.1 Overview of MapReduce-based Hamming-join

In this section, we introduce our implementation of the Hamming-join operation in MapReduce. As Figure 5 illustrates, the proposed algorithm includes three phases as explained below.

- **Preprocessing phase** Retrieve a sample from Datasets R and S. Then, use the sampled data to learn the hash function H. To handle data skew, build a data histogram for the sampled data and learn the data partitioning rule for the entire MapReduce job.
- **Global HA-Index building phase** Assume that the size of R is smaller than that of S. Partition R based on the pivot values from the data preprocessing step, then build the HA-Index for each partition using MapReduce by calling the H-Build function. Then, merge each local HA-Index to realize a global HA-Index for R.
- **Hamming-join phase** To join HA-Index of R with tuples in S, two possible options are applicable based on the size of R. More details are given later.

To learn the hash function, we utilize a random sample obtained from both R and S using reservoir sampling [22]. With the learned hash function H, high-dimensional data tuples in R and S are mapped into their corresponding binary codes. As discussed in the previous section, hash binary codes are ordered using the Gray order to preserve the clustering property. Hence, the data in each partition is more likely to share common $FLSSeq$ patterns. Then, we build the data histogram for the binary codes of the sampled data, and get a set of pivot values, denoted by *Pv*, for each Partition Pt_m. This guarantees that each partition receives approximately the same amount of data, where data in the various partitions is ordered according to the Gray order. More formally, given a set of data partitions Pt, and a set *Pv* of corresponding binary code values that form the partitioning pivots, Tuple $t_i \in Pt_m$, if the Gray order for t_i's binary code, say $\hat{U}_i$, belongs to the pivot range, i.e., $\hat{U}_i \in [Pv_m, Pv_{m+1})$, where Pv_m and Pv_{m+1} are the pivot values for Partition Pt_m.

Thus, let $|Pt_m|$ be the number of tuples belonging to Partition Pt_m, Pivot set *Pv* partition dataset R, s.t $R = \bigcup_{m=1}^{N} Pt_m$, and $|Pt_m| \simeq |Pt_{m+1}|$. Therefore, we can build the HA-Index and Hamming-join in each server as illustrated below.

5.2 Global HA-Index Building

Given the set of pivot values *Pv* selected in the preprocessing step, a MapReduce job partitions the data and builds an HA-Index locally in each partition. Specifically, before launching the map function, the selected pivots *Pv* and the learned hash function H are loaded into memory in each mapper via distributed cache in MapReduce. A mapper sequentially reads each input data tuple, say t_i, from the mapper's corresponding partition. The hash function maps the high-dimensional input data tuples into their corresponding binary codes, i.e., U. Then, a binary search is performed for the closest pivots in *Pv*. For the closest partition region, Partition ID is assigned. Finally, the mapper(s) produce(s) as output each object t_i along with its Partition ID, original dataset tuple identifier (R or S), and its binary code value U.

In the data shuffling phase, the key-value pairs emitted by all map functions are grouped by each distinct Partition ID, and a reduce function is called within each node. Each reduce function computes the local HA-Index via the H-Build function of Section 4, and produces the local HA-Index as output. In addition, a post-processing step to merge the various local HA-Indexes into one global HA-Index. Mainly, non-leaf nodes with the same $FLSSeq$ from the different local HA-Indexes are merged into one node, and the corresponding edges between the index nodes are relinked. Because the HA-Index is relatively small, the processing overhead is acceptable. After the first MapReduce job finishes, the global HA-Index for dataset R is built. This index is used by H-Search in the next phase.

5.3 Hamming-join

The second MapReduce job performs the Hamming-join in two possible ways.

Option(A): When Dataset R is small, i.e., storage of the leaf nodes of the HA-Index does not dominate the space of the HA-Index, the HA-Index maintains the leaf nodes as in Figure 3. Next, the Map function partitions Dataset S into N parts, i.e., $S = \bigcup_{i=1}^{N} S_i$. Then, it duplicates the global HA-Index for Dataset R and broadcasts to each server. The Map function computes the Hamming-join for Partition S_i and the replicated HA-Index of R. Specifically, before launching the MapReduce Job, the master node broadcasts the pivots *Pv*, the hash function H, and the global HA-Index of R to various servers. The main task of the mapper in the second MapReduce Job is to map high-dimensional data into binary codes, then partition dataset S into N partitions. Next, each reducer performs the Hamming-join between a pair of HA-Index and $\hat{S}_i$, and output the Hamming-join results.

Option(B): If Dataset R is big, e.g.,the number of tuples $|R|$ is more than millions, the storage of leaf nodes of the HA-Index dominates the space usage of the HA-Index. Therefore, the HA-Index of Dataset R does not maintain leaf nodes, and is duplicated to each server. By this way, the H-Search Algorithm 3 only returns the qualifying binary codes for Hamming-select, and a post-precessing step is carried out to find the tuple IDs for the qualifying binary codes. Take query tuple t_6 in Table 2a as an example. The H-Search algorithm computes binary codes from Table 2b, i.e., "101100010" and "101010010", which have a Hamming distance of 3 from t_6. In order to find the tuple IDs for those qualifying binaries, one post-processing step is invoked. Naturally, if Dataset R fits into memory, then the qualifying binaries are joined with R's hash table in memory. On the other hand, if Dataset R is too large to fit in memory, MapReduce hash-join [23] for Dataset R and the qualifying binaries is applied.

5.4 Shuffle Cost Analysis

The performance of MapReduce Hamming-join depends on the running time of Hamming-select as well as on the data shuffling cost. Let $|R| = m$ and $|S| = n$, respectively, d be the data dimension, and N be the number of partitions. In the previous work [4], Dataset R is duplicated and broadcast to each server, and the data shuffling cost is approximate to $O(mNd + nd)$. In this work, instead of duplicating the whole dataset R, only the HA-Index, is broadcast to each server. Hence, the data shuffling cost is reduced to $O(|HA|N + n)$, where $|HA|$ is the size of the HA-index. As introduced in Section 4, the space storage of HA is bound to $[O(\sqrt{m}), O(\sqrt[4]{m^3})]$. Therefore, the shuffling cost is bounded in $[O(\sqrt{m}N + n), O(\sqrt[m]{n^3}N + n)]$.

6. PERFORMANCE EVALUATION

We implement all the algorithms in Java. The experiments for Hamming-select are performed on an Intel(R) Xeon (R) E5320 1.86 GHz 4-core processor with 8G memory running Linux. The experiments on MapReduce are performed on a cluster of 16 nodes of Intel(R) Xeon (R) E5320 1.86 GHz 4-core machines with 8GB of main memory running Linux. We use Hadoop 0.22 and apply the default cluster environment setting. We evaluate the performance of the proposed techniques using the following three high-dimensional real datasets: (1) NUS-WIDE[2] is a web image dataset containing 269,648 images. We use 225-D block-wise color moments as the image features, thus obtaining a 225-dimension data. (2) Flickr[3] is a an image hosting website. We crawled 1 million images and extracted 512 features via the GIST Descriptor [24] (the data dimension is 512). (3) DBPedia[4] data aims to extract structured content from Wikipedia. We extract 1 million documents, and then apply standard NLP techniques to pre-process the documents, e.g., to remove stop words. We use the Latent Dirichlet Allocation (LDA) [25] model to extract topics, and we keep 250 topics for each document.

To evaluate the performance on larger data sizes, we synthetically generate more data while maintaining the same distribution as the original data distribution, e.g., as in [9, 10]. Suppose that the original dataset D has k dimensions. First, we get the frequencies of values in each dimension, and then sort the data in ascending order of their frequencies. Therefore, k copies of the dataset D are generated, one copy per dimension, e.g., D_j one copy of the dataset that is sorted based on the j-th dimension. Then, for each tuple, say t, in Dataset D, $t \in D$, we create a new tuple, say $\hat{t}$, according to the position of each component of t in the corresponding sorted copy D_j. For example, $t = (t_1, \ldots, t_j, \ldots, t_d)$ and t'_j is the first value larger than t_j in copy D_j, then $\hat{t} = (t'_1, \ldots, t'_j, \ldots, t'_d)$. If t_j is the largest element in Copy D_j, then $\hat{t}_j = t_j$. We use"$\times s$" to denote the increase in dataset size, where $s \in [5, 25]$ is the increase or scale factor. We consider the following approaches to evaluate Hamming-select:

(1) **Nested-Loops** is the naive approach to linearly XOR and count the binary data to perform the Hamming-distance computation. (2) **MultiHashTable** [4] is the state-of-the-art to search binary codes for similarity hashing that uses multiple-hash tables to reduce the linear search cost. While a large number of hash tables can achieve better performance, we limit ourselves to just 4 and 10 hash tables to avoid memory overflow. For short, we refer to these two possibilities, as MH-4 and MH-10. (3) **HEngine**^{s} [8] is the most recent work to improve the MultiHashTable approach in query time and memory usage. (4) **Radix-Tree** is the approach introduced in Section 4.2. (5) **Static HA-Index (SHA-Index) and Dynamic HA-Index (DHA-Index)** are the approaches introduced in Sections 4.3 and 4.4, respectively. SHA-Index(32) or DHA-Index(32) means that the length of the binary code is 32 bits.

We further evaluate the following approaches for kNN-select, and show how the approximate kNN-select can benefit from the enhancement of HA-Index searching over binary codes: (1) **Locality-Sensitive Hashing(E2LSH)** [18] is the state-of-the-art implementation for the data-independent LSH. We use 20 hash tables for E2LSH. (2) **LSB-TREE** [26] uses the Z-order curve to map high-dimensional data into one-dimensional Z-values, and index the Z-values using a B-tree. In our experiments, we build the LSB-Tree with 25 trees to compare the performance.

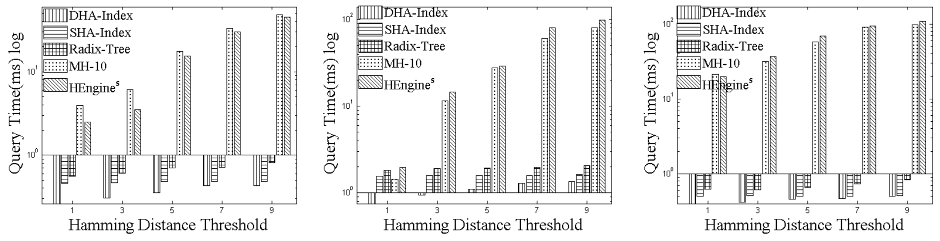
Also, we evaluate the following approaches to test the Self-Hamming-join, and verify how our approach of Map-Reduce Hamming-join can speedup the state-of-art algorithm for exact Self-kNN-join: (1) **Parallel-exact-KNN-join** (short as PGBJ) [10] is the state-of-the-art approach for performing exact kNN-join over multi-dimensional data in MapReduce, and it is 10 times speedup over the Z-order curve based approach [11]. We get the implementation generously provided by the authors [10]. (2) **Parallel Hamming-join via MultiHashTable** (PMH, for short) that handles approximate batch queries for web page duplicate identification [4]. PMH-10 means that 10 hash tables are used. (3) **Parallel Hamming-join via Dynamic HA-Index** (MRHA-Index, for short) is the approach introduced in Section 5. Specifically, in terms of the Hamming-join phase, if Option A is used, we term it MRHA-Index-A, and if Option B is used, we term it MRHA-Index-B.

The performance measures for each algorithm include the query time, the index update time, the index building time, memory usage, and the data shuffling cost. All performance measures are averaged over eight runs. Some running times are not plotted because they would use more than five hours. Unless mentioned, the default value of k is 50, and the Hamming-distance threshold $\hat{h}$ is 3. We choose the state-of-the-art Spectral Hashing [2] as the hash function in our experiments, but our approach is not limited to this hash function.

6.1 Results for Hamming-select

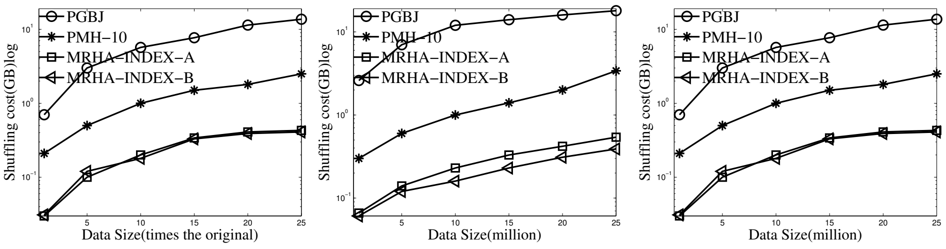
6.1.1 *Effectiveness of the HA-Index*

[2]http://lms.comp.nus.edu.sg/research/NUS-WIDE.htm
[3]htttp://www.flickr.com
[4]http://wiki.dbpedia.org/About



(a) NUS-WIDE (b) Flickr (c) DBPedia

Figure 6: Effect of the Hamming-distance threshold on Hamming select.



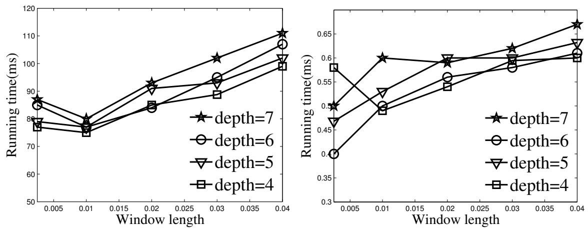
(a) NUS-WIDE (b) Flickr (c) DBPedia

Figure 7: Shuffling cost of Hamming-join and kNN-join.

Table 4 summarizes the query time, index update time, and memory space usage by the various approaches. Specifically, index update corresponds to the operation to delete one tuple first, then insert the same tuple back into the index. From Table 4, we have the following observations: 1) The Radix-Tree and HA-index-based approaches outperform the naive nested-loop and state-of-the-art methods [4, 8] on query time for the three datasets, mainly because the new proposed approach avoids many redundant Hamming-distance computations, and avoids scanning all the underlying data when they are hashed into the same bucket; 2) The HA-Index-based approach, i.e., the Static and Dynamic HA-Indexes, outperforms the Radix-Tree approach. The speedup is around 10 times because the Radix-Tree behaves as a prefix tree when many of the binary codes do not share long common prefixes, and hence cannot avoid the redundant Hamming distance computations; 3) The Static HA-Index shows better index-update time than that of the Dynamic HA-Index because the static segmentation enables us to track different binary segmentations directly, thus, we can search the paths of binary codes more efficiently; 4) The Radix-Tree and the HA-Index-based approaches save more memory than the state-of-the-art methods [4, 8] because the HA-Index-based approaches do not need to duplicate tuples and can share common $FLSS$s and $FLSSeq$s for different binary codes. This can reduce memory usage further; 5) For the Dynamic-HA-Index, if only the internal nodes of the HA-Index are kept, the memory usage can be reduced further. For instance, the memory usage for the Flickr and DBpedia datasets is reduced from 251MB and 225MB to 63MB and 47MB, respectively.

6.1.2 Effect of Hamming-Distance Threshold

We evaluate whether the running time of proposed approach is sensitive to the query threshold $\hat{h}$. Figure 6 gives the data query time when varying the Hamming-distance threshold. Notice that the query time of both the HA-Index-based approaches increases relatively slowly as the threshold increases. The reason is that the searching process in the HA-Index usually terminates early in the upper-level nodes, and this can improve the query speed. On the other hand, the searching path length of the Radix-Tree is not under control, and it tends to reach each leaf node when the Radix-Tree shares very little and changes to a prefix-tree-like format. However, state-of-the-art methods [4, 8] are sensitive to the Hamming-distance threshold because both approaches have to scan intermediate data to filter out non-qualifying tuples. Hence, the bigger $\hat{h}$ is, the more intermediate results that need to be scanned. This directly degrades the performance.



(a) Building Time (b) Query Processing Time.

Figure 8: DHA-Index building time and query processing when varying the window size.

6.1.3 Effect of HA-Index Parameters

We study the effects of the window length and the index depth of the dynamic HA-Index w.r.t. the index building and query processing times. The window length is normalized by the number of tuples in the dataset. Figure 8a illustrates that the building time for the HA-index drops as the depth decreases. The reason is that index construction stops early while the depth is small. Meanwhile, the HA-Index building time grows as the window size increases because the time to extract the same subpatterns for binaries of one window depends on the number of tuples inside the window. Meanwhile, the query processing time demonstrates stable growth as the

Table 4: Overall comparative study for Hamming-select: The dynamic-HA-Index is the most efficient in terms of query time and space usage, the binary code length is 32 bits. Notice for DHA-Index, **28/11** means 28MB and 11MB space usage for internal and leaf nodes were kept or only internal nodes, respectively.

	(a) NUS-WIDE			(b) Flickr			(c) DBPedia		
method	query time(ms)	update time(ms)	space usage	query time(ms)	update time(ms)	space usage	query time(ms)	update time(ms)	space usage
Nested-Loops	16.42	15.22	/	42.97	41.19	/	59.16	53.53	/
MH-4	6.22	0.21	475	16.09	0.60	712	40.28	0.45	819
MH-10	4.91	0.25	531	14.03	0.83	1204	34.46	0.64	1364
HEngine^s	3.53	0.45	210	14.75	1.14	820	36.91	1.91	763
Radix Tree	1.61	0.19	39	3.98	0.64	365	17.64	0.44	352
SHA-Index	0.87	**0.16**	29	1.75	**0.52**	254	3.54	**0.43**	239
DHA-Index	**0.68**	0.18	**28/11**	**0.74**	0.58	**251/63**	**1.07**	0.51	**225/47**

window size and index depth increase. Observe that the window size increases four times and the query processing time only grows by less than 10%. Thus, the HA-Index is not sensitive to these parameters.

6.1.4 *Comparison of Approaches for kNN-Select*

As introduced in Section 2, Hamming-select is a core operation for evaluating approximate kNN-select. In this section, we demonstrate the performance gains when using the HA-Index to speedup approximate kNN-select. Table 5 illustrates the runtime for data querying and index construction for LSH, LSB-Tree, and the HA-Index-based approaches. Observe that the HA-Index-based approach outperforms the state-of-the-art methods on all tasks when the binary code length is relatively large (i.e., 32 or 64 bits). Compared to the LSH approach, both HA-index-based approaches achieve two orders of magnitude speedup. The reason is that the LSH approach assumes uniformity in the distribution of the underlying data while real datasets are not uniform. In addition, the LSB-Tree can improve the query time compared to the LSH approach. However, the time to build the LSB-Tree index is expensive (more than 24 hours). In addition, the query and index building times for the HA-Index-based approach increases relatively smoothly as the binary code length increases. This demonstrates that the HA-Index approach is robust with the binary code length. Finally, the LSB-Tree consumes extensive disk space to store the index, LSB-Tree uses more than 20GB to store the index for the Flickr data, while the HA-Index-based approach only takes less than 300MB. This significantly reduces disk I/O time for the HA-Index-based approach.

6.2 Results of Hamming-join in MapReduce.

6.2.1 *Shuffling Cost*

We measure the effect of data size on the shuffling cost for PGBJ, PMH and the MRHA-Index. Figure 7 gives the data shuffle costs when the data size varies. The shuffle cost is plotted in logarithmic scale. The smaller the shuffle costs, the better the performance is. We observe that the shuffle costs for approximate kNN-join approach, i.e., PMH and MRHA-INDEX, are 10 times smaller when compared to the PGBJ approach. The reason is that the hashing technique maps the high-dimensional data into binary codes, and hence the data shuffling cost does not depend on the dimensions of the data. Notice that the data shuffling cost for PGBJ increases linearly with the data size. This is two orders of magnitude worse when compared to the data shuffling cost for the MRHA-INDEX approach. Duplicating and distributing the HA-Index into different nodes can improve the data shuffle cost 10 times less than that of

Table 5: Comparison with the state-of-the-art kNN-select approaches, when the dataset size is set to 300k tuples.

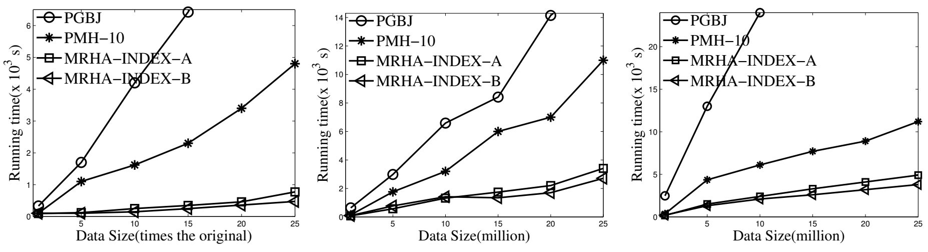
Dataset	Algorithm	Query time(ms)	Index build time
NUS-WIDE	LSH	2400	680(s)
	LSB-Tree(25)	47	37(Hr)
	SHA-Index(32)	2.74	68(s)
	SHA-Index(64)	4.78	97(s)
	DHA-Index(32)	1.64	87(s)
	DHA-Index(64)	2.43	103(s)
Flickr	LSH	340	1080(s)
	LSB-Tree(25)	63	50(Hr)
	SHA-Index(32)	2.21	176(s)
	SHA-Index(64)	3.54	189(s)
	DHA-Index(32)	2.17	210(s)
	DHA-Index(64)	2.88	244(s)
DBpedia	LSH	266	340(s)
	LSB-Tree(25)	59	44(Hr)
	SHA-Index(32)	2.94	150(s)
	SHA-Index(64)	4.88	290(s)
	DHA-Index(32)	2.18	230(s)
	DHA-Index(64)	3.85	310(s)

the PMH approach. On the other hand, the larger shuffle cost would stop the PGBJ approach from achieving a linear speedup and its corresponding execution time shows quadratic increase. The corresponding running times are given below. Finally, for the Hamming-join step in the HA-Index-based approach, Option B saves more data shuffling cost than Option A because the former does not need to duplicate the whole dataset into each server, and hence the space usage of the HA-Index remains relatively small.

6.2.2 *Scalability and Speedup*

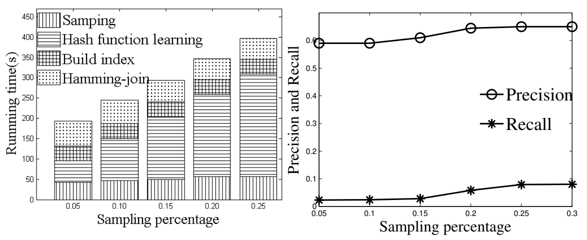
We investigate the scalability of the three approaches in Figure 9. The figure presents the results by varying the data size from 1 to 25 times of the original dataset sizes. From the figure, the overall execution time of PGBJ shows quadratic increase when the data size increases. For example, PGBJ's running time is almost 13 hours when the data is DBPedia$\times$15, which is excessively slow. The approximate kNN-join via similarity hashing always outperforms the PGBJ approach. Comparing with the state-of-the-art PMH-10 approach, the running time of the HA-Index outperforms PMH-10 by 5 times.

6.2.3 *Effect of Data Sampling*



(a) NUS-WIDE (b) Flickr (c) DBPedia

Figure 9: Speedup and scalability: Running time of Mapreduce Hamming-join and kNN-join.



(a) Query cost (b) Precision and Recall

Figure 10: Effect of sampling on query processing time, and precision/recall when varying the sampling data size.

Figure 10a gives the query execution time for the various processing phases of Hamming-join. From the Figure, more sampling of the data reflects the global data distribution more clearly, and this helps the sampling data pivot to partition different regions more evenly, and hence, improves the parallel HA-Index building and Hamming-join query time. The hash function learning usually takes more time, but for real-world applications, we only need to learn the hash function again when a certain amount of the new data is updated, which can save the time. Figure 10b illustrates how data sampling affects the query quality. Observe that the precision and recall can moderately improve as the sampling data size increases. However, the recall value is low.

7. CONCLUDING REMARKS

In this paper, we study the problem of efficiently performing the Hamming-select and Hamming-join operations. The proposed HA-Index approach executes the Hamming-distance-based similarity operations while avoiding unnecessary Hamming-distance computations. Extensive experiments using real datasets demonstrate that the proposed approaches outperforms the state-of-the-art techniques by two orders of magnitude. In future, it would be interest to explore hamming-distance similarity operation for relational operation i.e.,intersection [27].

8. REFERENCES

[1] J. Song, Y. Yang, Y. Yang, Z. Huang, and H. T. Shen, "Inter-media hashing for large-scale retrieval from heterogeneous data sources," ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 785–796.

[2] Y. Weiss, A. Torralba, and R. Fergus, "Spectral hashing," in *NIPS'08*, 2008, pp. 1753–1760.

[3] M. M. Bronstein, E. M. Bronstein, F. Michel, and N. Paragios, "Data fusion through crossmodality metric learning using similaritysensitive hashing," in *in Proc. CVPR*, 2010.

[4] G. S. Manku, A. Jain, and A. Das Sarma, "Detecting near-duplicates for web crawling," ser. WWW '07. New York, NY, USA: ACM, 2007, pp. 141–150.

[5] M. S. Charikar, "Similarity estimation techniques from rounding algorithms," in *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, ser. STOC '02. New York, NY, USA: ACM, 2002, pp. 380–388.

[6] M. Marvin and A. P. Seymour, "Perceptrons," *MIT Press*, 1969.

[7] D. Greene, M. Parnas, and F. Yao, "Multi-index hashing for information retrieval," in *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*, Nov 1994, pp. 722–731.

[8] A. Liu, K. Shen, and E. Torng, "Large scale hamming distance query processing," in *2011 IEEE 27th International Conference on Data Engineering (ICDE)*, April 2011, pp. 553–564.

[9] R. Vernica, M. J. Carey, and C. Li, "Efficient parallel set-similarity joins using mapreduce," ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 495–506.

[10] W. Lu, Y. Shen, S. Chen, and B. C. Ooi, "Efficient processing of k nearest neighbor joins using mapreduce," *Proc. VLDB Endow.*, vol. 5, no. 10, pp. 1016–1027, Jun. 2012.

[11] C. Zhang, F. Li, and J. Jestes, "Efficient parallel knn joins for large data in mapreduce," ser. EDBT '12. New York, NY, USA: ACM, 2012, pp. 38–49.

[12] H. Kllapi, B. Harb, and C. Yu, "Near neighbor join," in *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, March 2014, pp. 1120–1131.

[13] F. Gray, "Pulse code communication," in *U.S. Patent 2,632,058*, 1953.

[14] X. Zhang, J. Qin, W. Wang, Y. Sun, and J. Lu, "Hmsearch: An efficient hamming distance query processing algorithm," in *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, ser. SSDBM. New York, NY, USA: ACM, 2013, pp. 19:1–19:12.

[15] Y. N. Silva, W. G. Aref, P.-Å. Larson, S. Pearson, and M. H. Ali, "Similarity queries: their conceptual evaluation, transformations, and processing," *VLDB J.*, vol. 22, no. 3, pp. 395–420, 2013.

[16] C. Yu, B. C. Ooi, K.-L. Tan, and H. V. Jagadish, "Indexing the distance: An efficient method to knn processing," ser. VLDB '01, San Francisco, CA, 2001, pp. 421–430.

[17] R. Weber, H.-J. Schek, and S. Blott, "A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces," ser. VLDB '98, San Francisco, CA, 1998, pp. 194–205.

[18] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," *Commun. ACM*, vol. 51, no. 1, pp. 117–122, Jan. 2008.

[19] D. R. Morrison, "Patricia;practical algorithm to retrieve information coded in alphanumeric," *J. ACM*, vol. 15, no. 4, pp. 514–534, Oct. 1968.

[20] C. Faloutsos, "Multiattribute hashing using gray codes," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '86. ACM, 1986, pp. 227–238.

[21] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[22] J. S. Vitter, "Random sampling with a reservoir," *ACM Trans. Math. Softw.*, vol. 11, no. 1, pp. 37–57, Mar. 1985.

[23] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian, "A comparison of join algorithms for log processing in mapreduce," ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 975–986.

[24] A. Oliva and A. Torralba, "Modeling the shape of the scene: A holistic representation of the spatial envelope," *International Journal of Computer Vision*, vol. 42, pp. 145–175, 2001.

[25] A. K. McCallum, "Mallet: A machine learning for language toolkit," 2002, http://mallet.cs.umass.edu.

[26] Y. Tao, K. Yi, C. Sheng, and P. Kalnis, "Efficient and accurate nearest neighbor and closest pair search in high-dimensional space," *ACM Trans. Database Syst.*, vol. 35, no. 3, pp. 20:1–20:46, Jul. 2010.

[27] W. J. A. Marri, Q. M. Malluhi, M. Ouzzani, M. Tang, and W. G. Aref, "The similarity-aware relational intersect database operator," in *7th International Conference Similarity Search and Applications, SISAP*, 2014, pp. 164–175.

Joins for Hybrid Warehouses: Exploiting Massive Parallelism in Hadoop and Enterprise Data Warehouses

Yuanyuan Tian[1], Tao Zou[2]*, Fatma Özcan[1], Romulo Goncalves[3]†, Hamid Pirahesh[1]
[1]*IBM Almaden Research Center, USA* {ytian, fozcan, pirahesh}@us.ibm.com
[2]*Google Inc, USA* taozou@google.com
[3]*Netherlands eScience Center, Netherlands* r.goncalves@esciencecenter.nl

ABSTRACT

HDFS has become an important data repository in the enterprise as the center for all business analytics, from SQL queries, machine learning to reporting. At the same time, enterprise data warehouses (EDWs) continue to support critical business analytics. This has created the need for a new generation of special federation between Hadoop-like big data platforms and EDWs, which we call the *hybrid warehouse*. There are many applications that require correlating data stored in HDFS with EDW data, such as the analysis that associates click logs stored in HDFS with the sales data stored in the database. All existing solutions reach out to HDFS and read the data into the EDW to perform the joins, assuming that the Hadoop side does not have the efficient SQL support.

In this paper, we show that it is actually better to do most data processing on the HDFS side, provided that we can leverage a sophisticated execution engine for joins on the Hadoop side. We identify the best hybrid warehouse architecture by studying various algorithms to join database and HDFS tables. We utilize Bloom filters to minimize the data movement, and exploit the massive parallelism in both systems to the fullest extent possible. We describe a new *zigzag join* algorithm, and show that it is a robust join algorithm for hybrid warehouses which performs well in almost all cases.

1. INTRODUCTION

Through customer engagements, we observe that HDFS has become the *core storage system* for all enterprise data, including enterprise application data, social media data, log data, click stream data, and other Internet data. Enterprises are using various big data technologies to process this data and drive actionable insights. HDFS serves as the storage where other distributed processing frameworks, such as MapReduce [11] and Spark [43], access and operate on the large volumes of data.

At the same time, enterprise data warehouses (EDWs) continue to support critical business analytics. EDWs are usually shared-nothing parallel databases that support complex SQL processing, updates, and transactions. As a result, they manage up-to-date data and support various business analytics tools, such as reporting and dashboards.

Many new applications have emerged, requiring the access and correlation of data stored in HDFS and EDWs. For example, a company running an ad-campaign may want to evaluate the effectiveness of its campaign by correlating click stream data stored in HDFS with actual sales data stored in the database. These applications together with the co-existence of HDFS and EDWs have created the need for a new generation of special federation between Hadoop-like big data platforms and EDWs, which we call the *hybrid warehouse*.

Many existing solutions [37, 38] to integrate HDFS and database data use utilities to replicate the database data onto HDFS. However, it is not always desirable to empty the warehouse and use HDFS instead, due to the many existing applications that are already tightly coupled to the warehouse. Moreover, HDFS still does not have a good solution to update data in place, whereas warehouses always have up-to-date data. Other alternative solutions either statically pre-load the HDFS data [41, 17, 18], or fetch the HDFS data at query time into EDWs to perform joins [14, 13, 28]. They all have the implicit assumption that SQL-on-Hadoop systems do not perform joins efficiently. Although this was true for the early SQL-on-Hadoop solutions, such as Hive [39], it is not clear whether the same still holds for the current generation solutions such as IBM Big SQL [19], Impala [20], and Presto [33]. There was a significant shift last year in the SQL-on-Hadoop solution space, where these new systems moved away from MapReduce to shared-nothing parallel database architectures. They run SQL queries using their own long-running daemons executing on every HDFS DataNode. Instead of materializing intermediate results, these systems pipeline them between computation stages. In fact, the benefit of applying parallel database techniques, such as pipelining and hash-based aggregation, have also been previously demonstrated by some alternative big data platforms to MapReduce, like Stratosphere [5], Asterix [6], and SCOPE [10]. Moreover, HDFS tables are usually much bigger than database tables, so it is not always feasible to ingest HDFS data and perform joins in the database. Another important observation is that enterprises are investing more on big data systems like Hadoop, and less on expensive EDW systems. As a result, there is more capacity on the Hadoop side. Remotely reading HDFS data into the database introduces significant overhead and burden on the EDWs because they are fully utilized by existing applications, and hence carefully monitored and managed.

Split query processing between the database and HDFS has been addressed by PolyBase [13] to utilize vast Hadoop resources. HDFS clusters usually run on cheaper commodity hardware and have much larger capacity than databases. However, PolyBase only considers

*Work described in this paper was done while the author was working at IBM
†Work described in this paper was done while the author was working at IBM

10.5441/002/edbt.2015.33

pushing down limited functionality, such as selections and projections, and considers pushing down joins only when both tables are stored in HDFS.

Federation [21, 2, 34, 40, 30] is a solution to integrate data stored in autonomous databases, while exploiting the query processing power of all systems involved. However, existing federation solutions use a client-server model to access the remote databases and move the data. In particular, they use JDBC/ODBC interfaces for pushing down a maximal sub-query and retrieving its results. Such a solution ingests the result data serially through the single JDBC/ODBC connection, and hence is only feasible for small amounts of data. In the hybrid warehouse case, a new solution that connects at a lower layer is needed to exploit the massive parallelism on both the HDFS side and the EDW side.

In this paper, we identify an architecture for hybrid warehouses by 1-) building a system that provides parallel data movement by exploiting the massive parallelism of both HDFS and EDWs to the fullest extent possible, and 2-) studying the problem of efficiently executing joins between HDFS and EDW data. We start by adapting well-known distributed join algorithms, and propose extensions that work well between a parallel database and an HDFS cluster. Note that these joins work across two heterogeneous systems and hence have asymmetric properties that needs to be taken into account.

HDFS is optimized by large bulk I/O, and as a result record level indexing does not provide significant performance benefits. Other means, like column-store techniques [1, 31, 29], need to be exploited to speed up data ingestion.

Parallel databases use various techniques to optimize joins and minimize data movement. They use broadcast joins when one of the tables participating in the join is small enough, and the other is very large, to save communication cost. The databases also exploit careful physical data organization for joins. They rely on query workloads to identify joins between large tables, and co-partition them on the join key to avoid data communication at query time.

In the hybrid warehouse, these techniques have limited applicability. Broadcast joins can only be used in limited cases, because the data involved is usually very large. As the database and the HDFS are two independent systems that are managed separately, co-partitioning related tables is also not an option. As a result, we need to adapt existing join techniques to optimize the joins between very large tables when neither is partitioned on the join key. It is also very important to note that no existing EDW solution in the market today has a good solution for joining two large tables when they are not co-partitioned.

We exploit Bloom filters to reduce the data communication costs in joins for hybrid warehouses. A Bloom filter is a compact data structure that allows testing whether a given value is in a set very efficiently, with controlled false positive rate. Bloom filters have been proposed in the distributed relational query setting [25]. But they are not used widely, because they introduce overhead of extra computation and communication. In this paper, we show that Bloom filters are almost always beneficial when communicating data in the hybrid warehouse which integrated two heterogeneous and massively parallel data platforms, as opposed to the homogeneous parallel databases. Furthermore, we describe a new join algorithm, the *zigzag join*, which uses Bloom filters both ways to ensure that only the records that will participate in the join need to be transferred through the network. The zigzag join is most effective when the tables involved in the join do not have good local predicates to reduce their sizes, but the join itself is selective.

In this work, we consider executing the join both in the database and on the HDFS side. We implemented the proposed join algorithms for the hybrid warehouse using a commercial shared-nothing parallel database with the help of user-defined functions (UDFs), and our own execution engine for joins on HDFS, called JEN. To implement JEN, we took a prototype of the I/O layer and the scheduler from an existing SQL-on-Hadoop system, IBM Big SQL [19], and extended it with our own runtime engine which is able to pipeline operations and overlay network communication with processing and data scanning. We observe that with such a sophisticated execution engine on HDFS, it is actually often better to execute the joins on the HDFS side.

The contributions of this paper are summarized as follows:

- Through detailed experiments, we show that it is often better to execute joins on the HDFS side as the data size grows, when there is a sophisticated execution engine on the HDFS side. To the best of our knowledge, this is the first work that argues for such a solution.

- We describe JEN, a sophisticated execution engine on the HDFS side to fully exploit the various optimization strategies employed by a shared-nothing parallel database architecture, including multi-threading, pipelining, hash-based aggregation, etc. JEN utilizes a prototype of the I/O layer and the scheduler from IBM Big SQL and provides parallel data movement between HDFS and an EDW to exploit the massive parallelism on both sides.

- We revisit the join algorithms that are used in distributed query processing, and adapt them to work in the hybrid warehouse between two heterogeneous massively parallel data platforms. We utilize Bloom filters, which minimize the data movement and exploit the massive parallelism in both systems.

- We describe a new join algorithm, the zigzag join, which uses Bloom filters on both sides, and provide a very efficient implementation that minimizes the overhead of Bloom filter computation and exchange. We show that the zigzag join algorithm is a robust algorithm that performs the best for hybrid warehouses in almost all cases.

The rest of the paper is organized as follows: We start with a concrete example scenario, including our assumptions, in Section 2. The join algorithms are discussed in Section 3. We implemented our algorithms using a commercial parallel database and our own join execution engine on HDFS. In Section 4, we describe this implementation. We provide detailed experimental results in Section 5, discuss related work in Section 6, and conclude in Section 7.

2. AN EXAMPLE SCENARIO

In this paper, we study the problem of joins in the hybrid warehouse. We will use the following example scenario to illustrate the kind of query workload we focus on. This example represents a wide range of real application needs.

Consider a retailer, such as Walmart or Target, which sells products in local stores as well as online. All the transactions, either offline or online, are managed and stored in a parallel database, whereas users' online click logs are captured and stored in HDFS. The retailer wants to analyze the correlation of customers' online behaviors with sales data. This requires joining the transaction table T in the parallel database with the log table L on HDFS. One such analysis can be expressed as the following SQL query.

SELECT L.url_prefix, **COUNT**(*)
FROM T, L
WHERE T.category = 'Canon Camera'
AND region(L.ip)= 'East Coast'
AND T.uid=L.uid
AND T.tdate >= L.ldate **AND** T.tdate <= L.ldate+1
GROUP BY L.url_prefix

This query tries to find out the number of views of the urls visited by customers with IP addresses from East Coast who bought Canon cameras within one day of their online visits.

Now, we look at the structure of the example query. It has local predicates on both tables, followed by an equi-join. The join is also coupled with predicates on the joined result, as well as group-by and aggregation. In this paper, we will describe our algorithms using this example query.

In common setups, a parallel database is deployed on a small number (10s to 100s) of high-end servers, whereas HDFS resides on a large number (100s to 10,000s) of commodity machines. We assume that the parallel database is a full-fledged shared-nothing parallel database. It has an optimizer, indexing support and sophisticated SQL engine. On the HDFS side, we assume a scan-based processing engine without any indexing support. This is true for all the existing SQL-on-Hadoop systems, such as MapReduce-based Hive [39], Spark-based Shark [42], and Impala [20]. We do not tie the join algorithm descriptions to a particular processing framework, thus we generalize any scan-based distributed data processor on HDFS as a HQP (HDFS Query Processor). For data characteristics, we assume that both tables are large, but the HDFS table is much larger, which is the case in most realistic scenarios. In addition, since we focus on analytic workloads, we assume there is always group-by and aggregation at the end of the query. As a result, the final query result is relatively small. Finally, without loss of generality, we assume that queries are issued at the parallel database side and the final results are also to be returned at the database side. Note that forwarding a query from the database to HDFS is relatively cheap, so is passing the final results from HDFS back to the database.

Note that in this paper we focus on the actual join algorithms for hybrid warehouses, thus we only include a two-way join in the example scenario. Real big data queries may involve joining multiple tables. For these cases, we need to rely on the query optimizer in the database to decide on the right join orders, since queries are issued at the database side in our setting. However, the study of the join orders in hybrid warehouse is beyond the scope of this paper.

3. JOIN ALGORITHMS

In this section, we describe a number of algorithms for joining a table stored in a shared-nothing parallel database with another table stored in HDFS. We start by adapting well-known distributed join algorithms, and explore ways to minimize the data movement between these two systems by utilizing Bloom filters. While existing approaches [27, 26, 25, 24, 32] were designed for homogeneous environments, our join algorithms work across two heterogeneous systems in the hybrid warehouse. When we design the algorithms, we strive to leverage the processing power of both systems and maximize parallel execution.

Before we describe the join algorithms, let's provide a brief introduction to Bloom filters first. A Bloom filter is essentially a bit array of m bits with k hash functions defined to summarize a set of elements. Adding an element to the Bloom filter involves applying the k hash functions on the element and setting the corresponding positions of the bit array to 1. Symmetrically, testing whether an element belongs to the set requires simply applying the hash functions and checking whether all of the corresponding bit positions are set to 1. Obviously, the testing incurs some false positives. However, the false positive rate can be computed based on m, k and n, where n is the number of unique elements in the set. Therefore, m and k can be tuned for desired false positive rate. Bloom filter is a compact and efficient data structure for us to take advantage of the join selectivity. By building a Bloom filter on the join keys of one table, we can use it to prune out the non-joinable records from the other table.

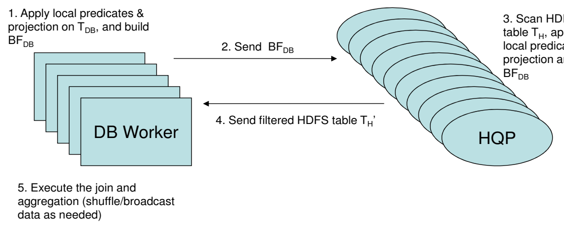

1. Apply local predicates & projection on T_{DB}, and build BF_{DB}
2. Send BF_{DB}
3. Scan HDFS table T_H, apply local predicates, projection and BF_{DB}
4. Send filtered HDFS table T_H'
DB Worker
HQP
5. Execute the join and aggregation (shuffle/broadcast data as needed)

Figure 1: Data flow of DB-side join with Bloom filter

3.1 DB-Side Join

Many database/HDFS hybrid systems, including Microsoft Polybase [13], Pivotal HAWQ [15], TeraData SQL-H [14], and Oracle Big Data SQL [28], fetch the HDFS table and execute the join in the database. We first explore this approach, which we call DB-side join. In the plain version, the HDFS side applies local predicates and projection, and sends the filtered HDFS table in parallel to the database. The performance of this join method is dependent on the amount of data that needs to be transferred from HDFS. Two factors determine this size: the selectivity of the local predicates over the HDFS table and the size of the projected columns.

Note that the HDFS table is usually much larger than the database table. Even if the local predicates are highly selective, the filtered HDFS table can still be quite large. In order to further reduce the amount of data transferred from HDFS to the parallel database, we introduce a Bloom filter on the join key of the database table after applying local predicates, and send the Bloom filter to the HDFS side. This technique enables the use of the join selectivity to filter out HDFS records that cannot be joined. This DB-side join algorithm is illustrated in Figure 1.

In this DB-side join algorithm, each parallel database node (DB worker in Figure 1) first computes the Bloom filter for their local partitions and then aggregate them into a global Bloom filter (BF_{DB}) by simply applying bitwise OR. We take advantage of the query optimizer of the parallel database. After the filtered HDFS data is brought into the database, it is joined with the database data using the join algorithm (broadcast or repartition) chosen by the query optimizer. Note that in the DB-side join, the HDFS data may need to be shuffled again at the database side before the join (e.g. if repartition join is chosen by the optimizer), because we do not have access to the partitioning hash function of the database.

In the above algorithm, there are different ways to send the database Bloom filter to HDFS and transmit the HDFS data to the database. Which approach works best depends on the network topology and the bandwidth. We defer the discussion of detailed implementation choices to Section 4.

3.2 HDFS-Side Broadcast Join

The second algorithm is called HDFS-side broadcast join, or simply broadcast join. This is the first algorithm that executes the

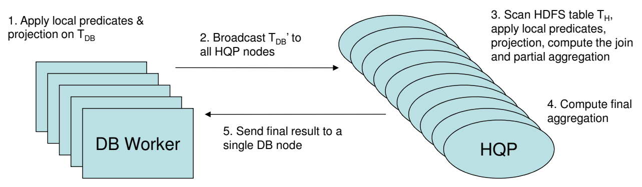

1. Apply local predicates & projection on T_{DB}
2. Broadcast T_{DB}' to all HQP nodes
3. Scan HDFS table T_H, apply local predicates, projection, compute the join and partial aggregation
4. Compute final aggregation
5. Send final result to a single DB node
DB Worker
HQP

Figure 2: Data flow of HDFS-side broadcast join

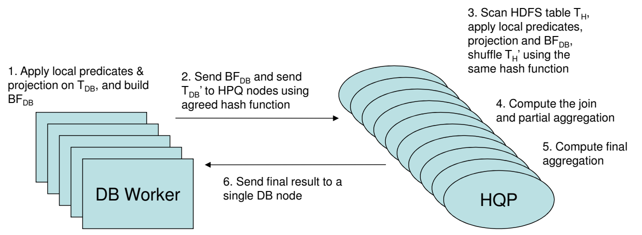

1. Apply local predicates & projection on T_{DB}, and build BF_{DB}
2. Send BF_{DB} and send T_{DB}' to HPQ nodes using agreed hash function
3. Scan HDFS table T_H, apply local predicates, projection and BF_{DB}, shuffle T_H' using the same hash function
4. Compute the join and partial aggregation
5. Compute final aggregation
6. Send final result to a single DB node
DB Worker
HQP

Figure 3: Data flow of HDFS-side repartition join with Bloom filter

join on the HDFS side. The rational behind this algorithm is that if the predicates on the database table are highly selective, the filtered database data is small enough to be sent to every HQP node, so that only local joins are needed without any shuffling of the HDFS data. When the join is executed on the HDFS side, it is logical to push-down the grouping and aggregation to the HDFS side as well. This way, only a small amount of summary data needs to be transferred back to the database to be returned to the user. The HDFS-side broadcast join algorithm is illustrated in Figure 2.

In the first step, each database node applies local predicates and projection over the database table. Each database node broadcasts its filtered partition to every HQP node (Step 2). Each HQP node performs a local join in Step 3. Group-by and partial aggregation are also carried out on the local data in this step. The final aggregation is computed in Step 4 and sent to the database in Step 5.

3.3 HDFS-Side Repartition Join

The second HDFS-side algorithm we consider is the HDFS-side repartition join, or simply repartition join. If the local predicates over the database table are not highly selective, then broadcasting the filtered data to all HQP nodes will not be a good option. In this case, we need a robust join algorithm. We expect the HDFS table to be much larger than the database table in practice, and hence it makes more sense to transfer the smaller database table and execute the final join at the HDFS side. Just as in the DB-side join, we can also improve this basic version of repartition join by introducing a Bloom filter. Figure 3 demonstrates this improved algorithm.

In Step 1, all database nodes apply local predicates over the database table, and project out the required columns. All database nodes also compute their local Bloom filters which are then aggregated into a global Bloom filter and sent to the HQP nodes. In this algorithm, the HDFS side and the database agree on the hash function to use when shuffling the data. In Step 2, all database nodes use this agreed hash function and send their data to the identified HQP nodes. This means that once the database data reaches the HDFS side, it doesn't need to be re-shuffled among the HQP nodes. In Step 3 of the HDFS-side repartition join, all HQP nodes apply the local predicates and projection over the HDFS table as well as the Bloom filter sent by the database. The Bloom filter further filters out the HDFS data. The HQP nodes use the same hash function to shuffle the filtered HDFS table. Then, they perform the join and partial aggregation (step 4). The final aggregation is executed on the HDFS side in Step 5 and sent to the database in Step 6.

3.4 HDFS-Side Zigzag Join

When local predicates on neither the HDFS table nor the database table are selective, we need to fully exploit the join selectivity to perform the join efficiently. In some sense, a selective join can be used as if it were extended local predicates on both tables. To illustrate this point, let's first introduce the concepts of *join-key selectivity* and *join-key predicate*.

Let T'_{DB} be the table after local predicates and projection on the database table T_{DB}, and T'_H be the table after local predicates and projection on the HDFS table T_H. We define $JK(T'_{DB})$ as the set of join keys in T'_{DB}, and $JK(T'_H)$ as the set of join keys in T'_H. We know that only the join keys in $JK(T'_{DB}) \cap JK(T'_H)$ will appear in the final join result. So, only $\frac{JK(T'_{DB}) \cap JK(T'_H)}{JK(T'_H)}$ fraction of the unique join keys in T'_H will participate in the join. We call this fraction the *join-key selectivity* on T'_H, denoted as $S_{T'_H}$. Likewise, the join-key selectivity on T'_{DB} is $S_{T'_{DB}} = \frac{JK(T'_{DB}) \cap JK(T'_H)}{JK(T'_{DB})}$. Leveraging the join-key selectivities through Bloom filters is essentially like applying extended local predicates on the join key columns of both tables. We call them *join-key predicates*.

Through the use of a 1-way Bloom filter, the DB-side join and the repartition join described in previous sections are only able to leverage the HDFS-side join-key predicate to reduce either the HDFS data transferred to the database or the HDFS data shuffled among the HQP workers. The DB-side join-key predicate is not utilized at all. Below, we introduce a new algorithm, zigzag join, to fully utilize the join-key predicates on both sides in reducing data movement, through the use of 2-way Bloom filters. Again, we expect the HDFS table to be much larger than the database table in practice, hence the final join in this algorithm is executed on the HDFS side, and both sides agree on the hash function to send data to the correct HQP nodes for the final join.

The zigzag join algorithm is described in Figure 4. In Step 1, all database nodes apply local predicates and projection, and compute their local Bloom filters. The database then computes the global Bloom filter BF_{DB} and sends it to all HQP nodes in Step 2. Like in the repartition join with Bloom filter, this Bloom filter helps reduce the amount of HDFS data that needs to be shuffled.

In Step 3, all HQP nodes apply their local predicates, projection and the database Bloom filter BF_{DB} over the HDFS table, and compute a local Bloom filter for the HDFS table. The local Bloom filters are aggregated into a global one, BF_H, which is sent to all database nodes. At the same time, the HQP nodes shuffle the filtered HDFS table based on the agreed hash function. In Step 5, the database nodes receive the HDFS Bloom filter BF_H and apply it to the database table to further reduce the number of database records that need to be sent. The application of Bloom filters on both sides ensure that only the data that will participate in the join (subject to false positive of the Bloom filter) needs to be transferred.

Note that in Step 5 the database data need to be accessed again. We rely on the advanced database optimizer to choose the best strategy: either to materialize the intermediate table $T_{DB'}$ after local predicates and projection are applied, or to utilize indexes to access the original table T_{DB}. It is also important to note that while the

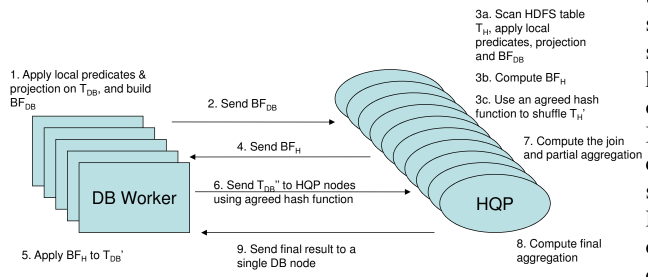

1. Apply local predicates & projection on T_{DB}, and build BF_{DB}
2. Send BF_{DB}
3a. Scan HDFS table T_H, apply local predicates, projection and BF_{DB}
3b. Compute BF_H
3c. Use an agreed hash function to shuffle T_H'
4. Send BF_H
5. Apply BF_H to T_{DB}'
6. Send T_{DB}'' to HQP nodes using agreed hash function
7. Compute the join and partial aggregation
8. Compute final aggregation
9. Send final result to a single DB node
DB Worker
HQP

Figure 4: Data flow of zigzag join

HDFS bloom filter is applied to the database data, the HQP nodes are shuffling the HDFS data in parallel, hence overlapping many steps of the execution.

In Step 6, the database nodes send the further filtered database data to the HQP nodes using the agreed hash function. The HQP nodes perform the join and partial aggregation (Step 7), collaboratively compute the global aggregation (Step 8), and finally send the result to the database (Step 9).

Note that zigzag join is the only join algorithm that can fully utilize the join-key predicates as well as the local predicates on both sides. The HDFS data shuffled across HQP nodes are filtered by the local predicates on T_H, the local predicates on T_{DB} (as BF_{DB} is built on T_{DB} after local predicates), and the join-key predicate on T'_H. Similarly, the database records transferred to the HDFS side are filtered by the local predicates on T_{DB}, the local predicates on T_H (as BF_H is built on T_H after local predicates), and the join-key predicate on T'_{DB}.

Although Bloom filters and semi-join techniques are known in the literature, they are not widely used in practice due to the overhead of computing Bloom filters and multiple data scans. However, the asymmetry of slow HDFS table scan and fast database table access makes these techniques more desirable in a hybrid warehouse. Note that a variant version of the zigzag join algorithm which executes the final join on the database side will not perform well, because scanning the HDFS table twice, without the help of indexes, is expected to introduce significant overhead.

4. IMPLEMENTATION

In this section, we provide an overview of our implementation of the join algorithms for the hybrid warehouse and highlight some important details.

4.1 Overview

In our implementation, we used IBM DB2 Database Partitioning Feature (DPF), which is a shared-nothing distributed version of DB2, as our EDW. We implemented all the above join algorithms using C user-defined functions (UDFs) in DB2 DPF and our own C++ MPI-based join execution engine on HDFS, called JEN. JEN is our specialized implementation of HQP used in the algorithm descriptions in Section 3. We used a propotype of the I/O layer and the scheduler from an early version of IBM Big SQL 3.0 [19], and build JEN on top of them. We also utilized Apache HCatalog [16] to store the meta data of the HDFS tables.

JEN consists of a single coordinator and a number of workers, with each worker running on an HDFS DataNode. JEN workers are responsible for reading parts of HDFS files, executing local query plans, and communicating with other workers, the coordinator, and DB2 DPF workers. Each JEN worker is multi-threaded, capable of exploiting all the cores on a machine. The communication between two JEN workers or with the coordinator is done through TCP/IP sockets. The JEN coordinator has multiple roles. First, it is responsible for managing the JEN workers and their state so that workers know which other workers are up and running in the system. Second, it serves as the central contact for the JEN workers to learn the IPs of the DB2 workers and vice versa, so that they can establish communication channels for data transfers. Third, it is also responsible for retrieving the meta data (HDFS path, input format, etc) for HDFS tables from HCatalog. Once the coordinator knows the path of the HDFS table, it contacts the HDFS NameNode to get the locations of each HDFS block, and evenly assigns the HDFS blocks to the JEN workers to read, respecting data locality.

At the DB2 side, we utilized the existing database query engine as much as possible. For the functionalities not provided, such as computing and applying Bloom filters, and different ways of transferring data to and from JEN workers, we implemented them using *unfenced* C UDFs, which provide performance close to built-in functions as they run in the same process as the DB2 engine. The communication between a DB2 DPF worker and a JEN worker is also through TCP/IP sockets. Note that to exploit the multi-cores on a machine we set up multiple DB2 workers on each machine of a DB2 DPF cluster, instead of one DB2 worker enabled with multi-core parallelism. This is mainly to simplify our C UDF implementations, as otherwise we have to deal with intra-process communications inside a UDF.

Each of the join algorithms is invoked by issuing a *single* query to DB2. With the help of UDFs, this single query executes the *entire* join algorithm: initiating the communication between the database and the HDFS side, instructing the two sides to work collaboratively, and finally returning the results back to the user.

4.1.1 The DB-Side Join Example

Let's use an example to illustrate how the database side and the HDFS side collaboratively execute a join algorithm. If we want to execute the example query in Section 2 using the DB-side join with Bloom filter, we submit the following SQL query to DB2.

```
with LocalFilter(lf) as (
select get_filter(max(cal_filter(uid))) from T
where T.category='Canon Camera'
group by dbpartitionnum(tid)
),
GlobalFilter(gf) as (
select * from
table(select combine_filter(lf) from LocalFilter)
where gf is not null
),
Clicks(uid, url_prefix, ldate) as (
select uid, url_prefix, ldate
from GlobalFilter,
table(read_hdfs('L', 'region(ip)= \'East Coast\'',
'uid, url_prefix, ldate', GlobalFilter.gf, 'uid'))
)
select url_prefix, count(*) from Clicks, T
where T.category='Canon Camera' and Clicks.uid=T.uid
and days(T.tdate)-days(Clicks.ldate)>=0
and days(T.tdate)-days(Clicks.ldate)<=1
group by url_prefix
```

In the above SQL query, we assume that the database table T is distributed across multiple DB2 workers on the `tid` field. The first sub query (`LocalFilter`) uses two scalar UDFs `cal_filter` and `get_filter` together to compute a Bloom filter on the local partition of each DB2 worker We enabled the two UDFs to execute in parallel, and the statement `group by dbpartitionnum(tid)` further makes sure that each DB2 worker computes the Bloom filter

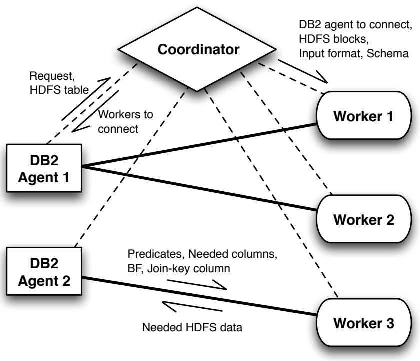


Figure 5: Communication in the `read_hdfs` UDF of the DB-side join with Bloom filter

on its local data in parallel. The second sub query (`GlobalFilter`) uses another scalar UDF `combine_filter` to combine the local Bloom filters into a single global Bloom filter (there is only one record which is the global Bloom filter returned for `GlobalFilter`). By declaring `combine_filter` "disallow parallel", we make sure it is executed once on one of the DB2 workers (all local Bloom filters are sent to a single DB2 worker). In the third sub query (`Clicks`), a table UDF `read_hdfs` is used to pass the following information to the HDFS side: the name of the HDFS table, the local predicates on the HDFS table, the projected columns needed to be returned, the global database Bloom filter, and the join-key column that the Bloom filter needs to be applied. In the same UDF, the JEN workers subsequently read the HDFS table and send the required data after applying predicates, projection and the Bloom filter back to the DB2 workers. The `read_hdfs` UDF is executed on each DB2 worker in parallel (the global Bloom filter is broadcast to all DB2 workers) and carries out the parallel data transfer from HDFS to DB2. After that, the join together with the group-by and aggregation is executed at the DB2 side. We pass a hint of the cardinality information to the `read_hdfs` UDF, so that the DB2 optimizer can choose the right plan for the join. The final result is returned to the user at the database side.

Now let's look into the details of the `read_hdfs` UDF. Since there is only one record in `GlobalFilter`, this UDF is called once per DB2 worker. When it is called on each DB2 worker, it first contacts the JEN coordinator to request for the connection information to the JEN workers. In return, the coordinator tells each DB2 worker which JEN worker(s) to connect to, and notifies the corresponding JEN workers to prepare for the connections from the DB2 workers. This process is shown in Figure 5. Without the loss of generality, let's assume that there are m DB2 workers and n JEN workers, and that $m \leq n$. For the DB-side join, the JEN coordinator evenly divides the n workers into m groups. Each DB2 worker establishes connections to all the workers in one group, as illustrated in Figure 5. After all the connections are established, each DB2 worker multi-casts the predicates on the HDFS table, the required columns from the HDFS table, the database Bloom filter and the join-key column to the corresponding group of JEN workers. At the same time, DB2 workers tell the JEN coordinator which HDFS table to read. The coordinator contacts the HCatalog to retrieve the paths of the corresponding HDFS files and the input format, and inquires the HDFS NameNode for the storage locations of the HDFS blocks. Then, the coordinator assigns the HDFS blocks and sends the assignment as well as the input format to the workers. After receiving all the necessary information, each JEN worker is

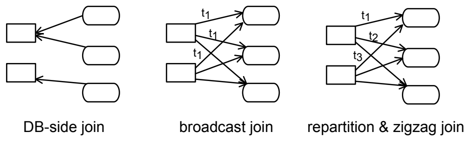


Figure 6: Data transfer patterns between DB2 workers and JEN workers in the join algorithms

ready to scan its share of the HDFS data. As it scans the data, it directly applies the local predicates and the Bloom filter from the database side, and sends the records with required columns back to its corresponding DB2 worker.

4.2 Locality-Aware Data Ingestion from HDFS

As our join execution engine on HDFS is scan-based, efficient data ingestion from HDFS is crucial for performance. We purposely deploy the JEN workers on all HDFS DataNodes so that we can leverage data locality when reading. In fact, when the JEN coordinator assigns the HDFS blocks to workers, it carefully considers the locations of each HDFS block to create balanced assignments and maximize the locality of data in a best-effort manner. Using this locality-aware data assignment, each JEN worker mostly reads data from local disks. We also enabled short-circuit reads for HDFS DataNodes to improve the local read speed. In addition, our data ingestion component uses multiple threads when multiple disks are used for each DataNode to further boost the data ingestion throughput.

4.3 Data Transfer Patterns

In this subsection, we discuss the data transfer patterns of different join algorithms. There are three types of data transfers that happen in all the join algorithms: among DB2 workers, among JEN workers, and between DB2 workers and JEN workers. For the data transfers among DB2 workers, we simply rely on DB2 to choose and execute the right transfer mechanisms. Among the JEN workers, there are three places that data transfers are needed: (1) shuffle the HDFS data for the repartition-based join in the repartition join (with/without Bloom filter) and the zigzag join, (2) aggregate the global HDFS Bloom filter for the zigzag join, and (3) compute the final aggregation result from the partial results on JEN workers in the broadcast join, the repartition join and the zigzag join. For (1), each worker simply maintains TCP/IP connections to all other workers and shuffles data through these connections. For (2) and (3), each worker sends the local results (either local Bloom filter or local aggregates) to a single designated worker chosen by the coordinator to finish the final aggregation.

The more interesting data transfers happen between DB2 workers and JEN workers. Again, there are three places that the data transfer is needed: shipping the actual data (HDFS or database), sending the Bloom filters, and transmitting the final aggregated results to the database for all the HDFS-side joins. Bloom filters and final aggregated results are much smaller than the actual data, how to transfer them has little impact on the overall performance. For the database Bloom filter sent to HDFS, we multi-cast the database Bloom filters to HDFS following the mechanism shown in Figure 5. For the HDFS Bloom filter sent to the database, we broadcast the HDFS Bloom filter from the designated JEN worker to all the DB2 workers. The final results on HDFS is simply transmitted from the designated JEN worker to a designated DB2 worker. In contrast to the above, we put more thoughts on how to ship the actual data between DB2 and HDFS. Figure 6 demonstrates the different patterns for transferring the actual data in the different join algorithms.

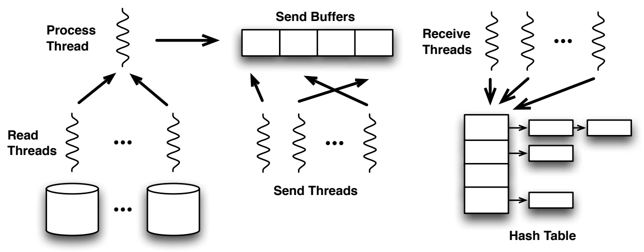


Figure 7: Interleaving of scanning, processing and shuffling of HDFS data in zigzag join

DB-side join with/without Bloom filter. For the two DB-side joins, we randomly partition the set of JEN workers into m roughly even groups, where m is the number of DB2 workers, then let each DB2 worker bring in the part of HDFS data in parallel from the corresponding group of JEN workers. DB2 can choose whatever algorithms for the final join that it sees fit based on data statistics. For example, when the database data is much smaller than HDFS data, the optimizer chooses to broadcast the database table for the join. When the HDFS data is much smaller than the database data, broadcasting the HDFS data is used. In the other cases, a repartition-based join algorithm is chosen. This means that when the HDFS data is transferred to the database side, it may need to be shuffled again among the DB2 workers. To avoid this second data transfer, we would have to expose the partitioning scheme of DB2 to JEN and teach the DB2 optimizer that the data received from JEN workers has already been partitioned in the desired way. Our implementation does not modify the DB2 engine, so we stick with this simpler and non-invasive data transfer scheme for the DB-side joins.

Broadcast join. There are multiple ways to broadcast the database data to JEN workers. One way is to let each DB2 worker connect to all the JEN workers and deliver its data to every worker. Another way is to have each DB2 worker only transfer its data to one JEN worker, which further passes on the data to all other workers. The second approach puts less stress on the inter-connection between DB2 and HDFS, but introduces a second round of data transfer among the JEN workers. We found empirically that broadcast join only works better than other algorithms when the database table after local predicates and projection is very small. For that case, even the first transfer pattern does not put much strain on the inter-connection between DB2 and HDFS. Furthermore, the second approach actually introduces extra latency because of the extra round of data transfer. For the above reasons, we use the first data transfer scheme in our implementation of the broadcast join.

Repartition join with/without Bloom filter and zigzag join. For these three join algorithms, the final join happens at the HDFS side. We expose the hash function for the final repartition-based join in JEN (DB2 workers can get this information from the JEN coordinator). When a database record is sent to the HDFS side, the DB2 worker uses the hash function to identify the JEN worker to send to directly.

4.4 Pipelining and Multi-Threading in JEN

In the implementation of JEN, we try to pipeline operations and parallelize computation as much as possible. Let's take the sophisticated zigzag join as an example.

At the beginning, every JEN worker waits to receive the global Bloom filter from DB2, which is a blocking operation, since all the remaining operations depend on this Bloom filter. After the Bloom filter is obtained, each worker starts to read its portion of the HDFS table (mostly from local disks) immediately. The data ingestion component is able to dedicate one *read thread* per disk when multiple disks are used for an HDFS DataNode. In addition, a separate *process thread* is used to parse the raw data into records based on the input format and schema of the HDFS table. Then it applies the local predicates, projection and the database Bloom filter on each record. For each projected record that passes all the conditions, this thread uses it to populate the HDFS-side Bloom filter, and applies the shuffling hash function on the join key to figure out which JEN worker this record needs to be sent to for the repartition-based join. Then, the record is put in a send buffer ready to be sent. All the above operations on a record are pipelined inside the process thread. At the same time, a pool of *send threads* poll the sending buffers to carry out the data transfers. Another pool of *receive threads* simultaneously receive records from other workers. And for each record received in a receive thread, it uses the record to build the hash table for the join. The multi-threading in this stage of the zigzag join is illustrated in Figure 7. As can be seen, scanning, processing and shuffling (sending and receiving) of HDFS data are carried out totally in parallel. In fact, the repartition join (with/without Bloom filter) also shares the similar interleaving of scanning, processing and shuffling of HDFS data. Note that reading from HDFS and shuffling data through networks are expensive operations, although we only have one process thread which applies the local predicates, Bloom filter and the projection, it is never the bottleneck.

As soon as the reading from HDFS finishes (read threads are done), a local Bloom filter is built on each worker. The workers send local Bloom filters to a designated worker to compute the global Bloom filter and pass it on to the DB2 workers. After that, every worker waits to receive and buffer the data from DB2 in the background. Once the local hash table is built (the send and receive threads in Figure 7 are all done), the received database records are used to probe the hash table, produce join results, and subsequently apply a hash-based group-by and aggregation immediately. Here again, all the operations on a database record are pipelined. When all the local aggregates are computed, each worker sends its partial result to a designated worker, which computes the final aggregate and sends to a single DB2 worker to return to the user.

Note that in our implementation of the zigzag join, we choose to build the hash table from the filtered HDFS data and use the transferred database data to prob the hash table for the final join, although the database data are expected to be smaller in most cases. This is because the filtered HDFS data is already being received during the scan of the HDFS table due to multi-threading. Empirically, we find that the receiving of the HDFS data is usually done soon after the scan is finished. On the other hand, the database data will not start to arrive until the HDFS table scan is done, as the HDFS side bloom filter is fully constructed only after all HDFS data are processed. Therefore, it makes more sense to start building the hash table on the filtered HDFS data while waiting for the later arrival of the database data. The current version of JEN requires that all data fit in memory for the local hash-based join on each worker. In the future, we plan to support spilling to disk to over come this limitation.

5. EXPERIMENTAL EVALUATION

Experimental Setup. For the HDFS cluster, we used 31 IBM System x iDataPlex dx340 servers. Each consisted of two quad-core Intel Xeon E5540 64-bit 2.8GHz processors (8 cores in total), 32GB RAM, 5x DATA disks and interconnected using 1Gbit Ethernet. Each server ran Ubuntu Linux (kernel version 2.6.32-24) and

Java 1.6. One server was dedicated as the NameNode, whereas the other 30 were used as DataNodes. We reserved 1 disk for the OS, and the remaining 4 for HDFS on each DataNode. HDFS replication factor is set to 2. A JEN worker was run on each DataNode and the JEN coordinator was run on the Namenode. For DB2 DPF, we used 5 servers. Each had 2x Intel Xeon CPUs @ 2.20GHz, with 6x physical cores each (12 physical cores in total), 12x SATA disks, 1x 10 Gbit Ethernet card, and a total of 96GB RAM. Each node runs 64-bit Ubuntu Linux 12.04, with a Linux Kernel version 3.2.0-23. We ran 6 database workers on each server, resulting in a total of 30 DB2 workers. 11 out of the 12 disks on each server were used for DB2 data storage. Finally, the two clusters were connected by a 20 Gbit switch.

Dataset. We generated synthetic datasets in the context of the example query scenario described in Section 2. In particular, we generated a transaction table T of 97GB with 1.6 billion records stored in DB2 DPF and a log table L on HDFS with about 15 billion records. The log table is about 1TB when stored in text format. We also stored the log table in the Parquet columnar format [31] with Snappy compression [36], to more efficiently ingest data from HDFS. The I/O layer of our JEN workers is able to push down projections when reading from this columnar format. The 1TB text log data is reduced to about 421GB in Parquet format. By default, our experiments were run on the Parquet formatted data, but in Section 5.4, we will compare Parquet format against text format to study their effect on performance. The schemas of the transaction table and the log table are listed below.

```
T(uniqKey bigint, joinKey int, corPred int, indPred
int, predAfterJoin date, dummy1 varchar(50), dummy2
int, dummy3 time)

L(joinKey int, corPred int, indPred int, predAfterJoin
date, groupByExtractCol varchar(46), dummy char(8))
```

The transaction table T is distributed on a unique key, called uniqKey, across the DB2 workers. The two tables are joined on a 4-byte int field joinKey. In both tables, there is one int column correlated with the join key called corPred, and another int column independent of the join key called indPred. They are used for local predicates. The date fields, named predAfterJoin, on the two tables are used for the predicate after the join. The varchar column groupByExtractCol in L is used for group-by. The remaining columns in each table are just dummy columns. Values of all fields in the two tables are uniformly distributed. The query that we ran in our experiments can be expressed in SQL as follows.

```
select extract_group(L.groupByExtractCol), count(*)
from T, L
where T.corPred<=a and T.indPred<=b
and L.corPred<=c and L.indPred<=d
and T.joinKey=L.joinKey
and days(T.predAfterJoin)-days(L.predAfterJoin)>=0
and days(T.predAfterJoin)-days(L.predAfterJoin)<=1
group by extract_group(L.groupByExtractCol)
```

In the above query, the local predicates on T and L are on the combination of the corPred and the indPred columns, so that we can change the join selectivities given the same selectivities of the combined local predicates. In particular, by modifying constants a and c, we can change the number of join keys participating in the final join from each table; but we can also modify the constants b and d accordingly so that the selectivity of the combined predicates stay intact for each table. We apply a UDF (extract_group) on the varchar column groupByExtractCol to extract an int column as the group-by column for the final aggregate count(*). To fully exploit

	HDFS tuples shuffled	DB tuples sent
repartition	5,854 million	165 million
repartition(BF)	591 million	165 million
zigzag	591 million	30 million

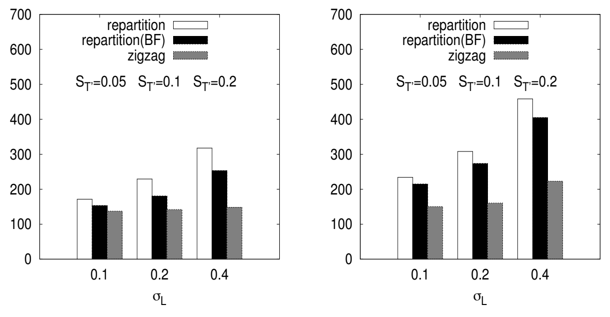
Table 1: Zigzag join vs repartition joins ($\sigma_T = 0.1$, $\sigma_L = 0.4$, $S_{L'} = 0.1$, $S_{T'} = 0.2$): # tuples shuffled and sent

the SQL support in DB2, we build one index on (corPred, indPred) and another index on (corPred, indPred, joinKey) of table T. The second index enables calculations of Bloom filters on T using an index-only access plan.

There are 16 million unique join keys in our dataset, so we create Bloom filters of 128 million bits (16MB) using 2 hash functions, which provides roughly 5% false positive rate. Note that exploring the different combinations of Bloom filter size and number of hash functions have been well studied before [9] and is beyond the scope of this paper. Our particular choice of the parameter values gave us good performance results in our experiments.

Our experiments were the only workloads that ran on the DPF cluster and the HDFS cluster. But, we purposely allocated less resources to the DPF cluster to mimic the case that the database is more heavily utilized. For all the experiments, we reported the warm-run performance numbers (we ran each experiments multiple times and excluded the first run when taking average).

In all the figures shown below, we denote the database table T after local predicates and projection as T' (predicate selectivity denoted as σ_T), and the HDFS table L after local predicates and projection as L' (predicate selectivity denoted as σ_L). We further represent the join-key selectivity on T' as $S_{T'}$ and the join-key selectivity on L' as $S_{L'}$.



(a) $\sigma_T = 0.1, S_{L'} = 0.1$ (b) $\sigma_T = 0.2, S_{L'} = 0.2$

Figure 8: Zigzag join vs repartition joins: execution time (sec)

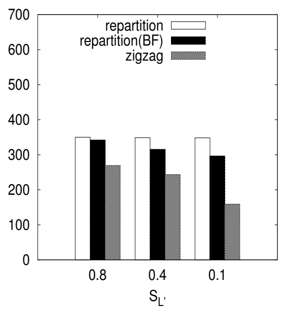
5.1 HDFS-Side Joins

We first study the HDFS-side join algorithms. We start with demonstrating the superiority of our zigzag join to the other reparation-based joins and then investigate when to use the broadcast join versus the repartition-based joins.

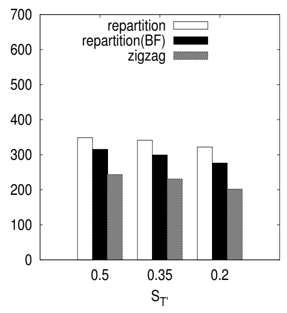
5.1.1 Zigzag Join vs Repartition Joins

We now compare the zigzag join to the repartition joins with and without Bloom filter. All three repartition-based join algorithms are best used when local predicate selectivities on both database and HDFS tables are low.

Figure 8 compares the execution times of the three algorithms with varying predicate and join-key selectivities on the Parquet formatted log table. It is evident that the zigzag join is the most efficient among the all repartition-based joins. It is up to 2.1x faster than the repartition join without Bloom filter and up to 1.8x faster than the repartition join with Bloom filter. When we zoom in the last three bars in Figure 8(a), Table 1 details the number of HDFS tuples shuffled across the JEN workers as well as the number of database tuples sent to the HDFS side for the three algorithms. The zigzag join is able to cut down the shuffled HDFS data by roughly 10x (corresponding to $S_{L'} = 0.1$) and the transferred database data by around 5x (corresponding to $S_{T'} = 0.2$). It is the only algorithm that can fully utilize the join-key predicates as well as the local predicates on both sides. In Figure 9, we fix the predicate selectivities $\sigma_T = 0.1$ and $\sigma_L = 0.4$ to explore the effect of different join-key selectivities $S_{L'}$ and $S_{T'}$ on the three algorithms. As expected, with the same size of T' and L', the performance of zigzag join improves with when the join-key selectivity $S_{L'}$ or $S_{T'}$ decreases.



(a) $S_{T'} = 0.5$



(b) $S_{L'} = 0.4$

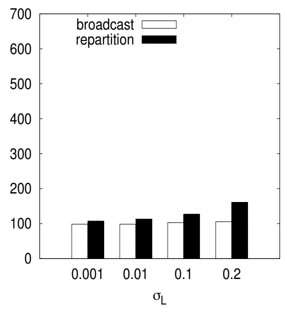
Figure 9: Zigzag join ($\sigma_T = 0.1$, $\sigma_L = 0.4$) with different $S_{L'}$ and $S_{T'}$ values: execution time (sec)

5.1.2 Broadcast Join vs Repartition Join

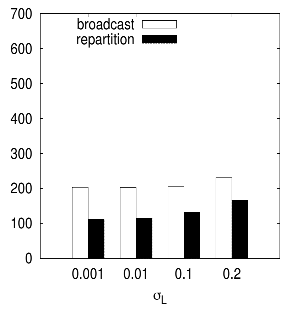
Besides the three repartition-based joins studied above, broadcast join is another HDFS-side join. To find out when this algorithm works best, we compare broadcast join and the repartition join without Bloom filter in Figure 10. We do not include the repartition join with Bloom filter or the zigzag join in this experiment, as even the basic repartition join is already comparable or better than broadcast join in most cases. The tradeoff between the broadcast join and the repartition join is basically broadcasting T' through the interconnection between the two clusters (the data transfered is 30×T' since we have 30 HDFS nodes) vs sending T' once through the interconnection and shuffling L' within the HDFS cluster. Due to the multi-threaded implementation described in Section 4.4, the shuffling of L' is interleaved with the reading of L in JEN, thus this shuffling overhead is somewhat masked by the reading time. As a result, broadcast join performs better only when T' is significantly smaller than L'. In our setting, broadcast join is only preferable when predicate on T is highly selective, e.g. $\sigma_T \leq 0.001$ (T' $\leq$ 25MB). In comparison, repartition-based joins are the more stable algorithms, and the zigzag join is the best HDFS-side algorithm in almost all cases.

5.2 DB-Side Joins

We now compare the DB-side joins with and without Bloom filter to study the effect of Bloom filter. As shown in Figure 11, Bloom filter is effective in most cases. For fixed local predicates on T (σ_T) and join-key selectivity on L' ($S_{L'}$), the benefit grows



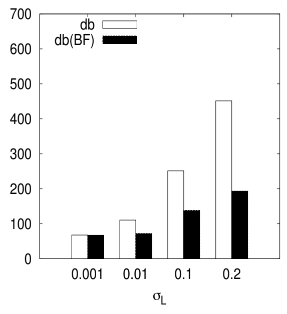
(a) $\sigma_T = 0.001$



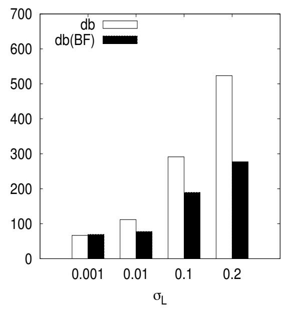
(b) $\sigma_T = 0.01$

Figure 10: Broadcast join vs repartition join: execution time (sec)

significantly as the size of L' increases. Especially for selective predicate on T, e.g. $\sigma_T = 0.05$, the impact of the Bloom filter is more pronounced. However, when the local predicates on L are very selective (σ_L is very small), e.g. $\sigma_L \leq 0.001$, the size of L' is already very small (e.g. less than 1GB when $\sigma_L = 0.001$), the overhead of computing, transferring and applying the Bloom filter can cancel out or even outweigh its benefit.



(a) $\sigma_T = 0.05, S_{L'} = 0.05$



(b) $\sigma_T = 0.1, S_{L'} = 0.1$

Figure 11: DB-side joins: execution time (sec)

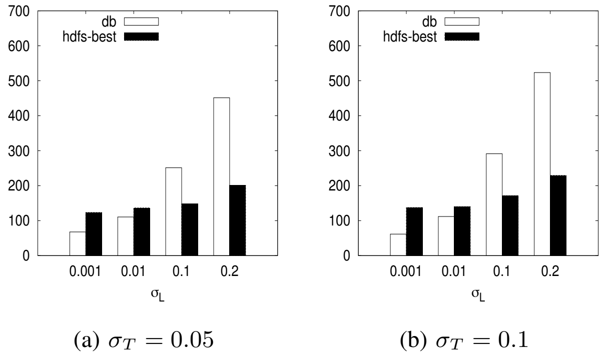
5.3 DB-Side Joins vs HDFS-Side Joins

Where to perform the final join, on the database side or the HDFS side, is a very important question that we want to address in this paper. Most existing solutions [13, 15, 14, 28] choose to always fetch the HDFS data and execute the join in the database, based on the assumption that SQL-on-Hadoop systems are slower in performing joins. Now, with the better designed join algorithms in this paper and the more sophisticated execution engine in JEN, we want to re-evaluate whether this is the right choice any more.

We start with the join algorithms without the use of Bloom filters, since the basic DB-side join is used in the existing database/HDFS hybrid systems, and the broadcast join and the basic repartition join are supported in most existing SQL-on-Hadoop systems. Figure 12 compares the DB-side join against the best of the HDFS-side joins (repartition join is the best for all cases in the figure). As shown in this figure, DB-side join performs better only when the predicate selectivity on the HDFS table is very selective ($\sigma_L \leq 0.01$). For lower selectivities, probably the common case, the repartition join shows very robust performance while the DB-side join very quickly deteriorates.

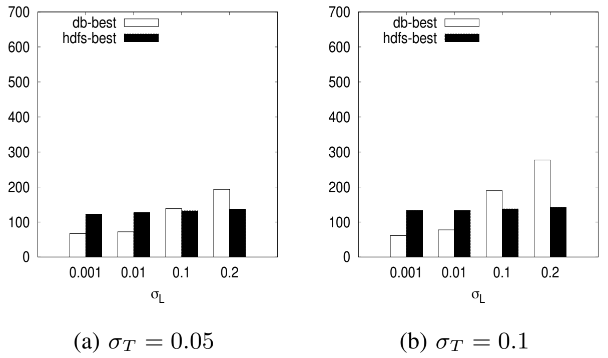
Now, let's also consider all the algorithms with Bloom filters and revisit the comparison in Figure 13. In most of the cases, the DB-side join with Bloom filter is the best DB-side join and zigzag join is the best HDFS-side join. Comparing this figure to Figure 12, the DB-side join still works better in the same cases as before, although all performance numbers are improved by the use of Bloom filters. The zigzag join shows very steady performance (execution time increases only slightly) with the increase of the L' size, in comparison with the steep deterioration rate of the DB-side join, making this HDFS-side join a very reliable choice for joins in the hybrid warehouse.

The above experimental results suggest that blindly executing joins in the database is not a good choice any more. In fact, for common cases when there is no highly selective predicate on the HDFS table, HDFS-side join is the preferred approach. There are several reasons for this. First of all, the HDFS table is usually much larger than the database table. Even with decent predicate selectivity on the HDFS table, the sheer size after predicates is still big. Second, as our implementation utilizes the DB2 optimizer as is, the HDFS data shipped to the database may need another round of data shuffling among the DB2 workers for the join. Finally, the database side normally has much less resources than the HDFS side, thus when both T' and L' are very large, HDFS-side join should be considered.



(a) $\sigma_T = 0.05$ (b) $\sigma_T = 0.1$

Figure 12: DB-side join vs HDFS-side join without Bloom filter: execution time (sec)

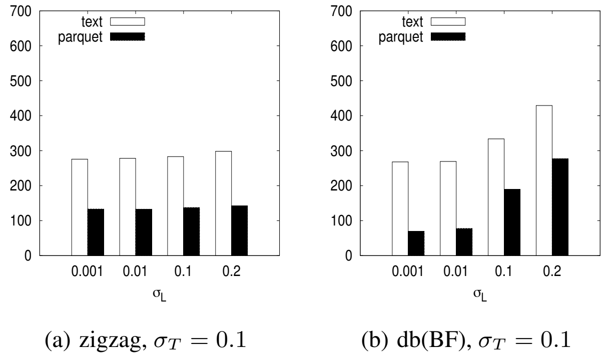


(a) $\sigma_T = 0.05$ (b) $\sigma_T = 0.1$

Figure 13: DB-side join vs HDFS-side join with Bloom filter: execution time (sec)

5.4 Parquet Format vs Text Format

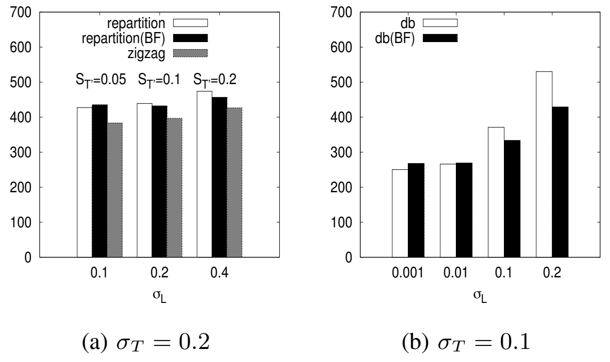
We now compare the join performance on the two different HDFS formats. We first pick the zigzag join, which is the best HDFS-side join, and the DB-side join with Bloom filter as the representatives, and show their performance on the Parquet and text formats in Figure 14.



(a) zigzag, $\sigma_T = 0.1$ (b) db(BF), $\sigma_T = 0.1$

Figure 14: Parquet format vs text format: execution time (sec)

Both algorithms run significantly faster on the Parquet format than on the text format. The 1TB text table on HDFS has already exceeded the aggregated memory size (960GB) of the HDFS cluster, thus simply scanning the data takes roughly 240 seconds in both cold and warm runs. After columnar organization and compression, the table is shrunk by about 2.4x, which can now well fit in the local file system cache on each DataNode. In addition, projection push-down can also be applied when reading from the Parquet format. Therefore, it only takes 38 seconds to read all the required fields from the Parquet data in a warm run. This huge difference in the scanning speed explains the big gap in the performance.



(a) $\sigma_T = 0.2$ (b) $\sigma_T = 0.1$

Figure 15: Effect of Bloom filter with text format: execution time (sec)

Next, we investigate the effect of using Bloom filter in joins on the text format. As shown in Figure 15, the improvement by Bloom filter is less dramatic on the text format than on the Parquet format. In some cases of the repartition join and the DB-side join, the overhead of computing, transferring and applying the Bloom filter even outweighs the benefit it brings. Again, the less benefit of Bloom filter is mainly due to the expensive scanning cost for the text format. In addition, there is another reason for the less effectiveness of Bloom filter in the repartition join and the zigzag join. Both algorithms utilize a database Bloom filter to reduce the amount of HDFS data to be shuffled, but with multi-threading, the shuffling is interleaved with the scan of HDFS data (see Section 4.4). For text format, the reduction of the shuffling cost is largely masked by the expensive scan cost, resulting in the less shown benefit. However, for the zigzag join, with a second Bloom filter to reduce the transferred database data, its performance is always robustly better.

5.5 Discussion

We now discuss the insights from our experimental study.

Among the HDFS-side joins, broadcast join only works for very limited cases, and even when it is better, the advantage is not dramatic. Repartition-based joins are the more robust solutions for HDFS-side joins, and the zigzag join with the 2-way Bloom filters always brings in the best performance.

Bloom filter also helps the DB-side join. However, with its steep deterioration rate, the DB-side join works well only when the HDFS table after predicates and projection is relatively small, hence its advantages are also confined to limited cases. For a large HDFS table without highly selective predicates, zigzag join is the most reliable join method that works the best most of the time, as it is the only algorithm that fully utilizes the join-key predicates as well as the local predicates on both sides.

HDFS data format significantly affects the performance of a join algorithm. Columnar format with fast compression and decompression techniques brings in dramatic performance boost, compared to the naive text format. So, when data needs to be accessed repeatedly, it is worthwhile to convert text format into the more advanced format.

Finally, we would like to point out that a major contribution to the nice performance of HDFS-side joins, is our sophisticated join execution engine on HDFS. It borrows the well-known runtime optimizations from parallel databases, such as pipelining and multi-threading. With our careful design in JEN, scanning HDFS data, network communication and computation are all fully executed in parallel.

6. RELATED WORK

In this paper, we study joins in the hybrid warehouse with two fully distributed and independent query execution engines in an EDW and an HDFS cluster, respectively. Although there has been rich literature on distributed join algorithms, most of these existing works study joins in a single distributed system.

In the context of parallel databases, Mackert and Lohman defined *Bloom join*, which uses Bloom filters to filter out tuples with no matching tuples in a join and achieves better performance than semijoin [25]. Michael et al showed how to use a Bloom filter based algorithm to optimize distributed joins where the data is stored in different sites [26]. In [12], DeWitt and Gerber studied join algorithms in a multiprocessor architecture and demonstrated that Bloom filter provides dramatic improvement for various join algorithms. PERF Join [24] reduces data transmission of two-way joins based on tuple scan order instead of using Bloom filters. It passes a bitmap of positions instead of a Bloom filter of values, in the second phase of semi-join. However, unlike Bloom join, it doesn't work well in parallel settings, when there are lots of duplicated values. Recently, Polychroniou et al proposed track join [32] to minimize network traffic for distributed joins by scheduling transfers of rows on a per join key basis. Determining the desired transfer schedule for each join key, however, requires a full scan of the two tables before the join. Clearly, for systems where scan is a bottleneck, track join would suffer from this overhead.

There has also been some work on join strategies in MapReduce [8, 3, 4, 23, 44]. Changchun et al. [44] presented several strategies to build the Bloom filter for the large dataset using MapReduce, and compared Bloom join algorithms of two-way and multi-way joins.

In this paper, we also exploit Bloom filters to improve distributed joins, but in a hybrid warehouse setting. Instead of one, our zigzag join algorithm uses two Bloom filters on both sides of the join to reduce the non-joining tuples. Two-way Bloom filters require scanning one of the tables two times, or materializing the intermediate result after applying local predicates. As a result, two-way Bloom filters are not as beneficial in a single distributed system. But, in our case we exploit the asymmetry between HDFS and the database, and scan the database table twice. Since HDFS scan is a dominating cost, scanning the database table twice, especially when we can leverage indexes, does not introduce significant overhead. As a result, our zigzag join algorithm provides robust performance in many cases.

With the need of hybrid warehouses, joins across shared-nothing parallel databases and HDFS have recently received significant attention. Most of the work either simply moves the database data to HDFS [37, 38], or moves the HDFS data to the database through bulk loading [38, 17], external tables [41, 17] or connectors [18, 38]. There are many problems with these approaches. First, HDFS tables are usually pretty big, so it is not always feasible to load them into the database. Second, such bulk reading of HDFS data into the database introduces an unnecessary burden on the carefully managed EDW resources. Third, database data gets updated frequently, but HDFS still does not support updates properly. Finally, all these approaches assume that the HDFS side does not have proper SQL support that can be leveraged.

Microsoft Polybase [13], Pivotal HAWQ [15], TeraData SQL-H [14], and Oracle Big Data SQL [28] all provide on-line approaches by moving only the HDFS data required for a given query dynamically into the database. They try to leverage both systems for query processing, but only simple predicates and projections are pushed down to the HDFS side. The joins are still evaluated entirely in the database. Polybase [13] considers split query processing, but joins are performed on the Hadoop side only when both tables are stored in HDFS.

Hadapt [7] also considers split query execution between the database and Hadoop, but the setup is very different. As it only uses single-node database severs for query execution, the two tables have to be either pre-partitioned or shuffled by Hadoop using the same hash function before the corresponding partitions can be joined locally on each database.

In this paper, we show that as the data size grows it is better to execute the join on the HDFS side, as we end up moving the smaller database table to the HDFS side.

Enabling the cooperation of multiple autonomous databases for processing queries has been studied in the context of federation [21, 2, 34, 40, 30] since the late 1970s. Surveys on federated database systems are provided in [35, 22]. However, the focus has largely been on schema translation and query optimization to achieve maximum query push down into the component databases. Little attention has been paid on the actual data movement between different component databases. In fact, many federated systems still rely on JDBC or ODBC connection to move data through a single data pipe. In the era of big data, even with maximum query push down, such naive data movement mechanisms result in serious performance issues, especially when the component databases are themselves massive distributed systems. In this paper, we provide parallel data movement by fully exploiting the massive parallelism between a parallel database and a join execution engine on HDFS to speed up the data movement when performing joins in the hybrid warehouse.

7. CONCLUSION

In this paper, we investigated efficient join algorithms in the context of a hybrid warehouse, which integrates HDFS with an EDW. We showed that it is usually more beneficial to execute the joins on the HDFS side, which is contrary to the current solutions which always execute joins in the EDW. We argue that the best hybrid warehouse architecture should execute joins where the bulk of the data is. In other words, it is better to move the smaller table to the side of the bigger table, whether it is in HDFS or in the database. This hybrid warehouse architecture requires a sophisticated execution engine on the HDFS side, and similar SQL capabilities on both sides. Given the recent advancement on SQL-on-Hadoop solutions [19, 20, 33], we believe this hybrid warehouse solution is now feasible. Finally, our proposed zigzag join algorithm, which performs joins on the HDFS side, utilizing Bloom filters on both sides, is the most robust algorithm that performs well in almost all cases.

8. REFERENCES

[1] D. Abadi, P. Boncz, and S. Harizopoulos. *The Design and Implementation of Modern Column-Oriented Database Systems*. Now Publishers Inc., 2013.

[2] S. Adali, K. S. Candan, Y. Papakonstantinou, and V. S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *SIGMOD*, pages 137–146, 1996.

[3] F. Afrati and J. Ullman. Optimizing joins in a map-reduce environment. In *EDBT*, pages 99–110, 2010.

[4] F. Afrati and J. Ullman. Optimizing multiway joins in a map-reduce environment. *TKDE*, 23(9):1282–1298, 2011.

[5] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke. The stratosphere platform for big data analytics. *VLDB Journal*, 23(6):939–964, 2014.

[6] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. Borkar, Y. Bu, M. Carey, K. Faraaz, E. Gabrielova, R. Grover, Z. Heilbron, Y.-S. Kim, C. Li, G. Li, J. M. Ok, N. Onose, P. Pirzadeh, V. Tsotras, R. Vernica, J. Wen, T. Westmann, I. Cetindil, and M. Cheelangi. AsterixDB: A scalable, open source BDMS. *PVLDB*, 7(14):1905–1916, 2014.

[7] K. Bajda-Pawlikowski, D. J. Abadi, A. Silberschatz, and E. Paulson. Efficient processing of data warehousing queries in a split execution environment. In *SIGMOD*, pages 1165–1176, 2011.

[8] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log processing in MapReduce. In *SIGMOD*, pages 975–986, 2010.

[9] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7), 1970.

[10] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2):1265–1276, 2008.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[12] D. J. DeWitt and R. H. Gerber. Multiprocessor hash-based join algorithms. In *VLDB '85*, pages 151–164, 1985.

[13] D. J. DeWitt, A. Halverson, R. V. Nehme, S. Shankar, J. Aguilar-Saborit, A. Avanes, M. Flasza, and J. Gramling. Split query processing in polybase. In *SIGMOD*, pages 1255–1266, 2013.

[14] Dynamic access: The SQL-H feature for the latest Teradata database leverages data in Hadoop. `http://www.teradatamagazine.com/v13n02/Tech2Tech/Dynamic-Access`.

[15] Pivotal HD: HAWQ. `http://www.gopivotal.com/sites/default/files/Hawq_WP_042313_FINAL.pdf`.

[16] HCatalog. `http://cwiki.apache.org/confluence/display/Hive/HCatalog`.

[17] High performance connectors for load and access of data from Hadoop to Oracle database. `http://www.oracle.com/technetwork/bdc/hadoop-loader/connectors-hdfs-wp-1674035.pdf`.

[18] IBM InfoSphere BigInsights. `http://pic.dhe.ibm.com/infocenter/bigins/v1r4/index.jsp`.

[19] IBM Big SQL 3.0: SQL-on-Hadoop without compromise. `http://public.dhe.ibm.com/common/ssi/ecm/en/sww14019usen/SWW14019USEN.PDF`.

[20] Impala. `http://github.com/cloudera/impala`.

[21] V. Josifovski, P. Schwarz, L. Haas, and E. Lin. Garlic: A new flavor of federated query processing for DB2. In *SIGMOD*, pages 524–532, 2002.

[22] D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.

[23] T. Lee, K. Kim, and H.-J. Kim. Join processing using bloom filter in MapReduce. In *RACS*, pages 100–105, 2012.

[24] Z. Li and K. A. Ross. PERF join: An alternative to two-way semijoin and bloomjoin. In *CIKM*, pages 137–144, 1995.

[25] L. F. Mackert and G. M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *VLDB*, page 149, 1986.

[26] L. Michael, W. Nejdl, O. Papapetrou, and W. Siberski. Improving distributed join efficiency with extended bloom filter operations. In *AINA*, 2007.

[27] J. K. Mullin. Optimal semijoins for distributed database systems. *TSE*, 16(5):558–560, 1990.

[28] Oracle Big Data SQL: One fast query, all your data. `https://blogs.oracle.com/datawarehousing/entry/oracle_big_data_sql_one`.

[29] The ORC format. `http://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.0.0.2/ds_Hive/orcfile.html`.

[30] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. D. Ullman. A query translation scheme for rapid implementation of wrappers. In *DOOD*, pages 161–186, 1995.

[31] Parquet. `http://parquet.io`.

[32] O. Polychroniou, R. Sen, and K. A. Ross. Track join: Distributed joins with minimal network traffic. In *SIGMOD*, pages 1483–1494, 2014.

[33] Presto. `http://prestodb.io`.

[34] M.-C. Shan, R. Ahmed, J. Davis, W. Du, and W. Kent. Pegasus: A heterogeneous information management system. In W. Kim, editor, *Modern Database Systems*, pages 664–682. ACM Press/Addison-Wesley Publishing Co., 1995.

[35] A. P. Sheth and J. A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Comput. Surv.*, 22(3):183–236, 1990.

[36] Snappy. `http://code.google.com/p/snappy`.

[37] Sqoop. `http://sqoop.apache.org`.

[38] Teradata connector for Hadoop. `http://developer.teradata.com/connectivity/articles/teradata-connector-for-hadoop-now-available`.

[39] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[40] A. Tomasic, L. Raschid, and P. Valduriez. Scaling access to heterogeneous data sources with DISCO. *TKDE*, 10(5):808–823, 1998.

[41] Teaching the elephant new tricks. `http://www.vertica.com/2012/07/05/teaching-the-elephant-new-tricks`.

[42] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and rich analytics at scale. In *SIGMOD*, pages 13–24, 2013.

[43] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[44] C. Zhang, L. Wu, and J. Li. Optimizing distributed joins with bloom filters using MapReduce. In *Computer Applications for Graphics, Grid Computing, and Industrial Environment*, pages 88–95, 2012.

Benchmarking Smart Meter Data Analytics

Xiufeng Liu, Lukasz Golab, Wojciech Golab and Ihab F. Ilyas
University of Waterloo, Canada
{xiufeng.liu,lgolab,wgolab,ilyas}@uwaterloo.ca

ABSTRACT

Smart electricity meters have been replacing conventional meters worldwide, enabling automated collection of fine-grained (every 15 minutes or hourly) consumption data. A variety of smart meter analytics algorithms and applications have been proposed, mainly in the smart grid literature, but the focus thus far has been on what can be done with the data rather than how to do it efficiently. In this paper, we examine smart meter analytics from a software performance perspective. First, we propose a performance benchmark that includes common data analysis tasks on smart meter data. Second, since obtaining large amounts of smart meter data is difficult due to privacy issues, we present an algorithm for generating large realistic data sets from a small seed of real data. Third, we implement the proposed benchmark using five representative platforms: a traditional numeric computing platform (Matlab), a relational DBMS with a built-in machine learning toolkit (PostgreSQL/MADLib), a main-memory column store ("System C"), and two distributed data processing platforms (Hive and Spark). We compare the five platforms in terms of application development effort and performance on a multi-core machine as well as a cluster of 16 commodity servers. We have made the proposed benchmark and data generator freely available online.

1. INTRODUCTION

Smart electricity grids, which incorporate renewable energy sources such as solar and wind, and allow information sharing among producers and consumers, are beginning to replace conventional power grids worldwide. Smart electricity meters are a fundamental component of the smart grid, enabling automated collection of fine-grained (usually every 15 minutes or hourly) consumption data. This enables dynamic electricity pricing strategies, in which consumers are charged higher prices during peak times to help reduce peak demand. Additionally, smart meter data analytics, which aims to help utilities and consumers understand electricity consumption patterns, has become an active area in research and industry. According to a recent report, utility data analytics is already a billion dollar market and is expected to grow to nearly 4 billion dollars by year 2020 [16].

A variety of smart meter analytics algorithms have been proposed, mainly in the smart grid literature, to predict electricity consumption and enable accurate planning and forecasting, extract consumption profiles and provide personalized feedback to consumers on how to adjust their habits and reduce their bills, and design targeted engagement programs to clusters of similar consumers. However, the research focus has been on the insights that can be obtained from the data rather than performance and programmer effort. Implementation details were omitted, and the proposed algorithms were tested on small data sets. Thus, despite the increasing amounts of available data and the increasing number of potential applications[1], it is not clear how to build and evaluate a practical and scalable system for smart meter analytics. This is exactly the problem we study in this paper.

1.1 Contributions

We begin with a *benchmark* for comparing the performance of smart meter analytics systems. Based on a review of the related literature (more details in Section 2), we identified four common tasks: 1) understanding the variability of consumers (e.g., by building histograms of their hourly consumption), 2) understanding the thermal sensitivity of buildings and households (e.g., by building regression models of consumption as a function of outdoor temperature), 3) understanding the typically daily habits of consumers (e.g., by extracting consumption trends that occur at different times of the day regardless of the outdoor temperature) and 4) finding similar consumers (e.g., by running times series similarity search). These tasks involve aggregation, regression and time series analysis. Our benchmark includes a representative algorithm from each of these four sets.

Second, since obtaining smart meter data for research purposes is difficult due to privacy concerns, we present a *data generator* for creating large realistic smart meter data sets from a small seed of real data. The real data set we were able to obtain consists of only 27,000 consumers, but our generator can create much larger data sets and allows us to stress-test the candidate systems.

Third, we implement the proposed benchmark using five state-of-the-art platforms that represent recent data management trends, including in-database machine learning, main-memory column stores, and distributed analytics. The five platforms are:

1. Matlab: a numeric computing platform with a high-level language;

2. PostgreSQL: a traditional relational DBMS, accompanied by MADLib [17], an in-database machine learning toolkit;

[1]See, e.g., a recent competition sponsored by the United States Department of Energy to create new apps for smart meter data: http://appsforenergy.challengepost.com.

3. "System C": a main-memory column-store commercial system (the licensing agreement does not allow us to reveal the name of this system);

4. Spark [28]: a main-memory distributed data processing platform;

5. Hive [25]: a distributed data warehouse system built on top of Hadoop, with an SQL-like interface.

We report performance results on our real data set and larger realistic data sets created by our data generator. Our main finding is that System C performs extremely well on our benchmark at the cost of the highest programmer effort: System C does not come with built-in statistical and machine learning operators, which we had to implement from scratch in a non-standard language. On the other hand, MADLib and Matlab make it easy to develop smart meter analytics applications, but they do not perform as well as System C. In cluster environments with very large data sizes, we found Hive easier to use than Spark and not much slower. Spark and Hive are competitive with System C in terms of efficiency (throughput per server) for several of the workloads in our benchmark.

Our benchmark (i.e., the data generator and the tested algorithms) is freely available for download at https://github.com/xiufengliu. Due to privacy issues, we are unable to share the real data set or the large synthetic data sets based upon it. However, a smart meter data set has recently become available at the Irish Social Science Data Archive[2] and may be used along with our data generator to create large publicly available data sets for benchmarking purposes.

1.2 Roadmap

The remainder of this paper is organized as follows. Section 2 summarizes the related work; Section 3 presents the smart meter analytics benchmark; Section 4 discusses the data generator; Section 5 presents our experimental results; and Section 6 concludes the paper with directions for future work.

2. RELATED WORK

2.1 Smart Meter Data Analytics

There are two broad areas of research in smart meter data analytics: those which use whole-house consumption readings collected by conventional smart meters (e.g., every hour) and those which use high-frequency consumption readings (e.g., one per second) obtained using specialized load-measuring hardware. We focus on the former in this paper, as these are the data that are currently collected by utilities.

For whole-house smart meter data feeds, there are two classes of applications: consumer and producer-oriented. Consumer-oriented applications provide feedback to end-users on reducing electricity consumption and saving money (see, e.g., [10, 21, 24]). Producer-oriented applications are geared towards utilities, system operators and governments, and provide information about consumers such as their daily habits for the purposes of load forecasting and clustering/segmentation (see, e.g., [1, 3, 5, 8, 12, 13, 14, 15, 22, 23]).

From a technical standpoint, both of the above classes of applications perform two types of operations: extracting representative features (see, e.g., [8, 10, 13, 14]) and finding similar consumers based on the extracted features (see, e.g., [1, 12, 23, 24, 26]). Household electricity consumption can be broadly decomposed into the temperature-sensitive component (i.e., the heating and cooling load) and the temperature-insensitive component (other appliances). Thus, representative features include those which measure the effect of outdoor temperature on consumption [4, 10, 23] and those which identify consumers' daily habits regardless of temperature [1, 8, 13], as well as those which measure the overall variability (e.g., consumption histograms) [3]. Our smart meter benchmark, which will be described in Section 3, includes four representative algorithms for characterizing consumption variability, temperate sensitivity, daily activity and similarity to other consumers.

We also point out recent work on smart meter data quality (specifically, handling missing data) [18], symbolic representation of smart meter time series [27], and privacy (see, e.g., [2]). These important issues are orthogonal to smart meter analytics, which is the focus of this paper.

2.2 Systems and Platforms for Smart Meter Data Analytics

Traditional options for implementing smart meter analytics include statistical and numeric computing platforms such as R and Matlab. As for relational database systems, two important technologies are main-memory databases, such as "System C" in our experiments, and in-database machine learning, e.g., PostgreSQL/MADLib [17]. Finally, a parallel data processing platform such as Hadoop or Spark is an interesting option for cluster environments. We have implemented the proposed benchmark in systems from each of the above classes (details in Section 5).

Smart meter analytics software is currently offered by several database vendors including SAP[3] and Oracle/Data Raker[4], as well as startups such as Autogrid.com, C3Energy.com and OPower.com. However, it is not clear what algorithms are implemented by these systems and how.

There has also been some recent work on efficient retrieval of smart meter data stored in Hive [20], but that work focuses on simple operational queries rather than the deep analytics that we address in this paper.

2.3 Benchmarking Data Analytics

There exist several database (e.g., TPC-C, TPC-H and TPC-DS) and big data[5] benchmarks, but they focus mainly on the performance of relational queries (and/or transactions) and therefore are not suitable for smart meter applications. Benchmarking time series data mining was discussed in [19]. Different implementations of time series similarity search, clustering, classification and segmentation were evaluated. While some of these operations are relevant to smart meter analytics, there are other important tasks such as extracting consumption profiles that were not evaluated in [19]. Additionally, [19] evaluated standalone algorithms whereas we evaluate data analytics platforms. Furthermore, [7] benchmarked data mining operations for power system analysis. However, its focus was on analyzing voltage measurements from power transmission lines, not smart meter data, and therefore the tested algorithms were different from ours. Finally, Arlitt et al. propose a benchmark for smart meter analytics that focuses on routine computations such as finding top customers and calculating monthly bills [9]. In contrast our work aims to discover more complex patterns in energy data. Their workload generator uses a Markov chain model that must be trained using a real data set.

[2]http://www.ucd.ie/issda/data/commissionforenergyregulationcer/

[3]http://www.sap.com/pc/tech/in-memory-computing-hana/software/smart-meter-analytics/index.html

[4]http://www.oracle.com/us/products/applications/utilities/meter-data-analytics/index.html

[5]https://amplab.cs.berkeley.edu/benchmark

We also note that the TCP benchmarks include the ability to generate very large synthetic databases, and there has been some research on synthetic database generation (see, e.g., [11]), but we are not aware of any previous work on generating realistic smart meter data.

3. THE BENCHMARK

In this section, we propose a performance benchmark for smart meter analytics. The primary goal of the benchmark is to measure the running time of a set of tasks that will be defined shortly. The input consists of n time series, each corresponding to one electricity consumer, in one or more text files. We assume that each time series contains hourly electricity consumption measurements (in kilowatt-hours, kWh) for a year, i.e., $365 \times 24 = 8760$ data points. For each consumption time series, we require an accompanying external temperature time series, also with hourly measurements.

For each task, we measure the running time on the input data set, both with a cold start (working directly from the raw files) and a warm start (working with data loaded into physical memory). In this version of the benchmark, we do not consider the cost of updates, e.g., adding a day's worth of new points to each time series. However, adding updates to the benchmark is an important direction for future work as read-optimized data structures that help improve running time may be expensive to update.

Utility companies may have access to additional data about their customers, e.g., location, square footage of the home or family size. However, this information is usually not available to third-party applications. Thus, the input to our benchmark is limited to the smart meter time series and publicly-available weather data.

We now discuss the four analysis tasks included in the proposed benchmark.

3.1 Consumption Histograms

The first task is to understand the variability of each consumer. To do this, we compute the distribution of hourly consumption for each consumer via a histogram. The x-axis in the histogram denotes various hourly consumption ranges and the y-axis is the frequency, i.e., the number of hours in the year whose electricity consumption falls in the given range. For concreteness, in the proposed benchmark we specify the histograms to be equi-width (rather than equi-depth) and we always use ten buckets.

3.2 Thermal Sensitivity

The second task is to understand the effect of outdoor temperature on the electricity consumption of each household. The simplest approach is to fit a least-squares regression line to the consumption-temperature scatter plot. However, in climates with a cold winter and warm summer, electricity consumption rises when the temperature drops in the winter (due to heating) and also rises when the temperature rises in the summer (due to air conditioning). Thus, a piecewise linear regression model is more appropriate.

We selected the recently-proposed algorithm from [10] for the benchmark, to which we refer as the 3-line algorithm. Consider a consumption-temperature scatter plot for a single consumer shown in Figure 1 (the actual points are not shown, but a point on this plot would correspond to a particular hourly consumption value and the outdoor temperature at that hour). The upper three lines correspond to the piecewise regression lines computed only for the points in the 90th percentile for each temperature value and the lower three lines are computed from the points in the 10th percentile for each temperature value. Thus, for each time series, the algorithm starts by computing the 10th and 90th percentiles for each temperature

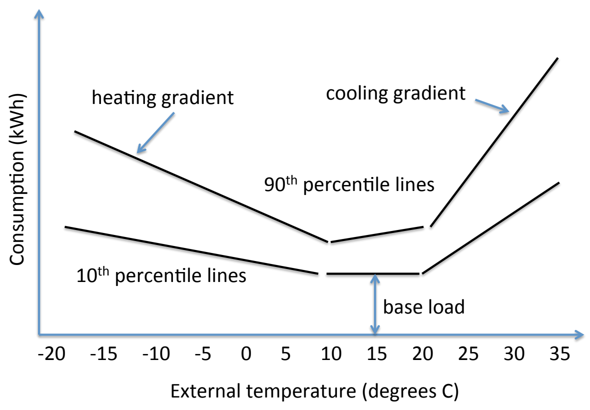


Figure 1: Example of the 3-line regression model.

value and then computes the two sets of regression lines. In the final step, the algorithm ensures that the three lines are not discontinuous and therefore it may need to adjust the lines slightly.

As shown in Figure 1, the 3-line algorithm extracts useful information for customer feedback. For instance, the slopes (gradients) of the left and right 90th percentile lines correspond to the heating and cooling sensitivity, respectively. A high cooling gradient might indicate an inefficient air conditioning system or a low air conditioning set point. Additionally, the height at the lowest point on the 10th percentile lines indicates *base load*, which is the electricity consumption of appliances and devices that are always on regardless of the temperature (e.g., a refrigerator, a dehumidifier, or a home security system).

3.3 Daily Profiles

The third task is to extract daily consumption trends that occur regardless of the outdoor temperature. For this, we use the periodic autoregression (PAR) algorithm for time series data from [8, 13]. The idea behind this algorithm is illustrated in Figure 2. At the top, we show a fragment of the hourly consumption time series for some consumer over a period of several days. We are only given the total hourly consumption, but the goal of the algorithm is to determine, for each hour, how much load is temperature-independent and how much additional load is due to temperature (i.e., heating or cooling). Once this is determined, the algorithm computes the average temperature-independent consumption at each hour of the day, illustrated at the bottom of Figure 2. Thus, for each consumer, the output consists of a vector of 24 numbers, denoting the expected consumption at different hours of the day due solely to the occupants' daily habits and not affected by temperature.

For each consumer and each hour of the day, the PAR algorithm fits an auto-regressive model, which assumes that the electricity consumption at that hour of the day is a linear combination of the consumption at the same hour over the previous p days (we use $p = 3$, as in [8]) and the outdoor temperature. Thus, it groups the input data set by consumer and by hour, and computes the coefficients of the auto-regressive model for each group.

3.4 Similarity Search

The final task is to find groups of similar consumers. Customer segmentation is important to utilities so they can determine how many distinct groups of customers there are and design targeted energy-saving campaigns for each group. Rather than choosing

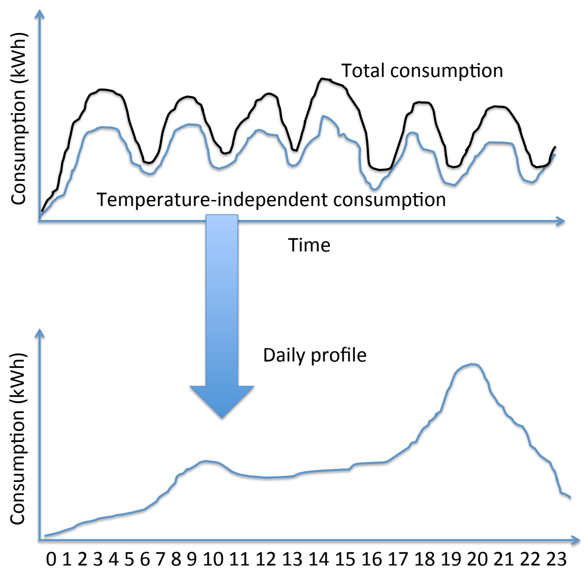


Figure 2: Example of a daily profile.

a specific clustering algorithm for the benchmark, we include a more general task: for each of the n time series given as input, we compute the top-k most similar time series (we use $k = 10$). The similarity metric we use is *cosine similarity*. Let X and Y be two time series. The cosine similarity between them is defined as their dot product divided by the product of their vector lengths, i.e, $\frac{X \cdot Y}{||X||*||Y||}$.

3.5 Discussion

To recap, the proposed benchmark consists of 1) the consumption histogram, 2) the 3-line algorithm for understanding the effect of external temperature on consumption, 3) the periodic autoregression (PAR) algorithm to extract typical daily profiles and 4) the time series similarity search to find similar consumers. The first three algorithms analyze the electricity consumption of each household in terms of its distribution, its temperature sensitivity and its daily patterns. The fourth algorithm finds similarities among different consumers. While many more smart meter analytics algorithms have been proposed, we believe the four tasks we have chosen accurately represent a variety of fundamental computations that might be used to extract insights from smart meter data.

In terms of computational complexity, the first three algorithms perform the same task for each consumption time series and therefore can be parallelized easily, while similarity search has quadratic complexity with respect to the number of time series. Computing histograms requires grouping the time series according to consumption values. The 3-line algorithm additionally requires grouping the data by temperature, and requires statistical operators such as quantiles and least-squares regression lines. The PAR and similarity algorithms require time series operations. Thus, the proposed benchmark tests the ability to extract different segments of the data and run various statistical and time series operations.

4. THE DATA GENERATOR

Recall from Section 3 that the proposed benchmark requires n time series as input, each corresponding to an electricity consumer. Testing the scalability of a system therefore requires running the benchmark with increasing values of n. Since it is difficult to obtain large amounts of smart meter data due to privacy issues, and since using randomly-generated time series may not give accurate results, we propose a data generator for realistic smart meter data.

The intuition behind the data generator is as follows. Since electricity consumption depends on external temperature and daily activity, we start with a small seed of real data and we generate the daily activity profiles (recall Figure 2) and temperature regression lines (recall Figure 1) for each consumer therein. To generate a new time series, we take the daily activity pattern from a randomly-selected consumer in the real data set, the temperature dependency from another randomly-selected consumer, and we add some white noise. Thus, we first disaggregate the consumption time series of existing consumers in the seed data set, and we then re-aggregate the different pieces in a new way to create a new consumer. This gives us a realistic new consumer whose electricity usage combines the characteristics of multiple existing consumers.

Figure 3 illustrates the proposed data generator. As a pre-processing step, we use the PAR algorithm from [13] to generate daily profiles for each consumer in the seed data set. We then run the k-means clustering algorithm (for some specified value of k, the number of clusters) to group consumers with similar daily profiles. We also run the 3-line algorithm and record the heating and cooling gradients for each consumer.

Now, creating a new time series proceeds as follows. We randomly select an activity profile cluster and use the cluster centroid to obtain the hourly consumption values corresponding to daily activity load. Next, we randomly select an individual consumer from the chosen cluster and we obtain its cooling and heating gradients. We then need to input a temperature time series for the new consumer and we have all the information we need to create a new consumption time series[6]. Each hourly consumption measurement of the new time series is generated by adding together 1) the daily activity load for the given hour, 2) the temperature-dependent load computed by multiplying the heating or cooling gradient by the given temperature value at that hour, and 3) a Gaussian white noise component with some specified standard deviation σ.

5. EXPERIMENTAL RESULTS

This section presents our experimental results. We start with an overview of the five platforms in which we implemented the proposed benchmark (Section 5.1) and a description of our experimental environment (Section 5.2). Section 5.3 then discusses our experimental findings using a single multi-core server, including the effect of data layout and partitioning (Section 5.3.1), the relative cost of data loading versus query execution (Section 5.3.2), and the performance of single-threaded and multi-threaded execution (Section 5.3.3 and 5.3.4, respectively). In Section 5.4, we investigate the performance of Spark and Hive on a cluster of 16 worker nodes. We conclude with a summary of lessons learned in Section 5.5.

5.1 Benchmark Implementation

We first introduce the five platforms in which we implemented the proposed benchmark. Whenever possible, we use native statistical functions or third-party libraries. Table 1 shows which functions were included in each platform and which we had to implement ourselves.

The baseline system is Matlab, a traditional numeric and statistical computing platform that reads data directly from files. We use

[6]In our experiments, we used the temperature time series corresponding to the southern-Ontario city from which we obtained the real data set.

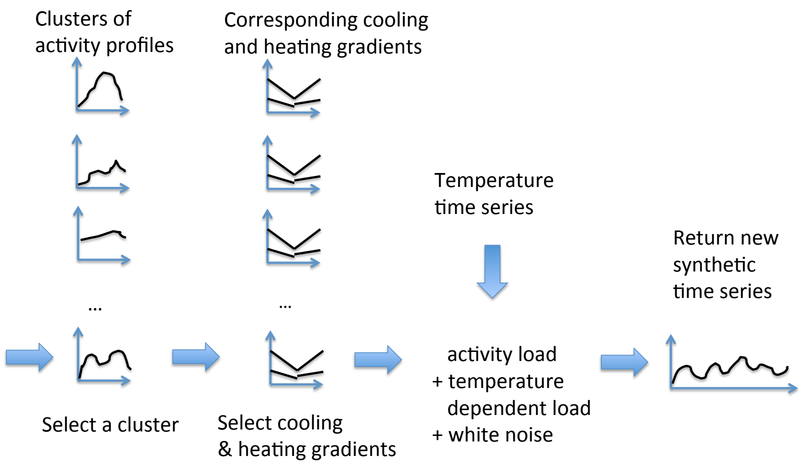

Clusters of activity profiles
Corresponding cooling and heating gradients
Temperature time series
Return new synthetic time series
...
...
Select a cluster
Select cooling & heating gradients
activity load + temperature dependent load + white noise

Figure 3: Illustration of the proposed data generator.

Table 1: Statistical functions built into the five tested platforms

Function	Matlab	MADLib	System C	Spark	Hive
Histogram	yes	yes	no	no	yes
Quantiles	yes	yes	no	no	no
Regression and PAR	yes	yes	no	third party library	third party library
Cosine Similarity	no	no	no	no	no

the built-in histogram, quantile, regression and PAR functions, and we implemented our own (very simple) cosine similarity function by looping through each time series, computing its similarity to every other time series, and, for each time series, returning the top 10 most similar matches.

We also evaluate PostgreSQL 9.1 and MADLib version 1.4 [17], which is an open-source platform for in-database machine learning. As we will explain later in this section, we tested two ways of storing the data: one measurement per row, and one customer per row with all the measurements for this customer stored in an array. Similarly to Matlab, everything we need except cosine similarity is built-in. We implemented the benchmark in PL/PG/SQL with embedded SQL, and we call the statistical functions directly from within SQL queries. We use the default settings for PostgreSQL[7].

Next, we use System C as an example of a state-of-the-art commercial system. It is a main-memory column store geared towards time series data. System C maps tables to main memory to improve I/O efficiency. In particular, at loading time, all the files are memory mapped to speed up subsequent data access. However, System C does not include a machine learning toolkit, and therefore we implemented all the required statistical operators as user-defined functions in the procedural language supported by it.

We also use Spark [28] as an example of an open-source distributed data processing platform. Spark reports improved performance on machine learning tasks over standard Hadoop/MapReduce due to better use of main memory [28]. We use the *Apache Math* library for regression, but we had to implement our own histogram, quantile and cosine similarity functions. We use the Hadoop Distributed File System (HDFS) as the underlying file system for Spark.

Finally, we test another distributed platform, Hive [25], which is built on top of Hadoop and includes a declarative SQL-like interface. Hive has a built-in histogram function, and we use *Apache Math* for regression. We implemented the remaining functions (quantiles and cosine similarity) in Java as user-defined functions (UDFs). The data are stored in Hive external tables.

In terms of programming time to implement our benchmark, PostgreSQL/MADLib required the least effort, followed by Matlab and Hive, while Spark and especially System C required by far the most effort. In particular, we found Hive UDFs easier to write than Spark programs. However, since we did not conduct a user study, these programmer effort observations should be treated as anecdotal.

In the remainder of this section, we will refer to the five tested platforms as Matlab, MADLib, C (or System C), Spark and Hive.

5.2 Experimental Environment

We run each of the four algorithms in the benchmark using each of the five platforms discussed above, and measure the running times and memory consumption. We use the following two testing environments.

- Our server has an Intel Core i7-4770 processor (3.40GHz, 4 Cores, hyper-threading is enabled, two hyper-threads per core), 16GB RAM, and a Seagate hard drive (1TB, 6 GB/s, 32 MB Cache and 7200 RPM), running Ubuntu 12.04 LTS with 64bit Linux 3.11.0 kernel. PostgreSQL 9.1 is installed with the settings "shared _buffers= 3072MB, temp_ buffers= 256MB, work_ mem=1024MB, checkpoint_segments =64" and default values for other configuration parameters.

- We also use a dedicated cluster with one administration node and 16 worker nodes. The administration node is the master node of Hadoop and HDFS, and clients submit jobs there. All the nodes have the same configuration: dual-socket Intel(R) Xeon(R) CPU E5-2620 (2.10GHz, 6 cores per socket, and two hyper-threads per core), 60GB RAM, running 64bit Linux with kernel version 2.6.32. The nodes

[7]We also experimented with turning off concurrency control and write-ahead-logging which are not needed in our application, but the performance improvement was not significant.

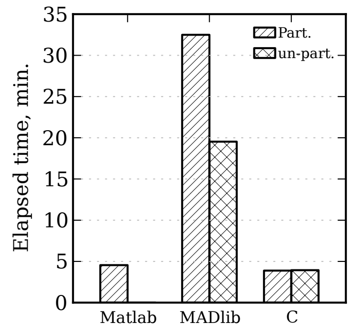


Figure 4: Data loading times, 10GB real dataset.

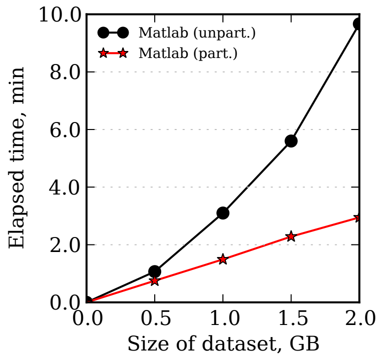


Figure 5: Impact of data partitioning on analytics, 3-line algorithm.

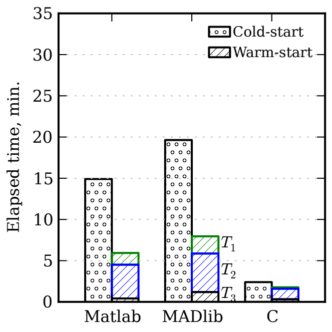


Figure 6: Cold-start vs. warm-start, 3-line algorithm, 10GB real dataset.

are connected via gigabit Ethernet, and a working directory is NFS-mounted on all the nodes.

Our real data set consists of $n = 27,300$ electricity consumption time series, each with hourly readings for over a year. We also obtained the corresponding temperature time series. The total data size is roughly 10 GB. We also use the proposed data generator to create larger synthetic data sets of size up to one Terabyte (which corresponds to over two million time series), and experiment with them in Section 5.4.

5.3 Single-Server Results

We begin by comparing Matlab, MADLib and System C running on a single multi-core server, using the real 10GB dataset.

5.3.1 *Data Loading and File Partitioning*

First, we investigate the effect of loading and processing one large file containing all the data versus one file per consumer. Figure 4 shows the time it took to load our 10-GB real data set into the three systems tested in this section, both in a partitioned (one file per consumer, abbreviated "part.") and non-partitioned (one big file, abbreviated "un-part.") format. The partitioned data load also includes the cost of splitting the data into small files. The loading time into PostgreSQL is the slowest of the three systems, but it is more efficient to bulk-load one large CSV file than many smaller files. System C is not significantly affected by the number of files. Matlab does not actually load any data and instead reads from files directly. The single bar reported for Matlab, of roughly 4.5 minutes, simply corresponds to the time it took to split the data set into small files.

Once data are loaded into tables in PostgreSQL or System C, the number of input files no longer matters. However, Matlab reads data directly from files, so the goal of our next experiment is to investigate the performance of analytics in Matlab given the two partitioning strategies discussed above. Figure 5 shows the running time of the 3-line algorithm using Matlab on (partitioned and non-partitioned) subsets of our real data sets sized from 0.5 to 2 GB. (We observed similar trends when running the other algorithms in the benchmark). The impact on Matlab is significant: it operates much more efficiently if each consumer's data are in a separate file. Upon further investigation, we noticed that Matlab reads the entire large file into an index which is then used to extract individual consumers' data; this is slower than reading small files one-by-one and running the 3-line algorithm on each file directly.

Based on the results of this experiment, in the remainder of this section, we always run Matlab with one file per consumer.

5.3.2 *Cold Start vs. Warm Start*

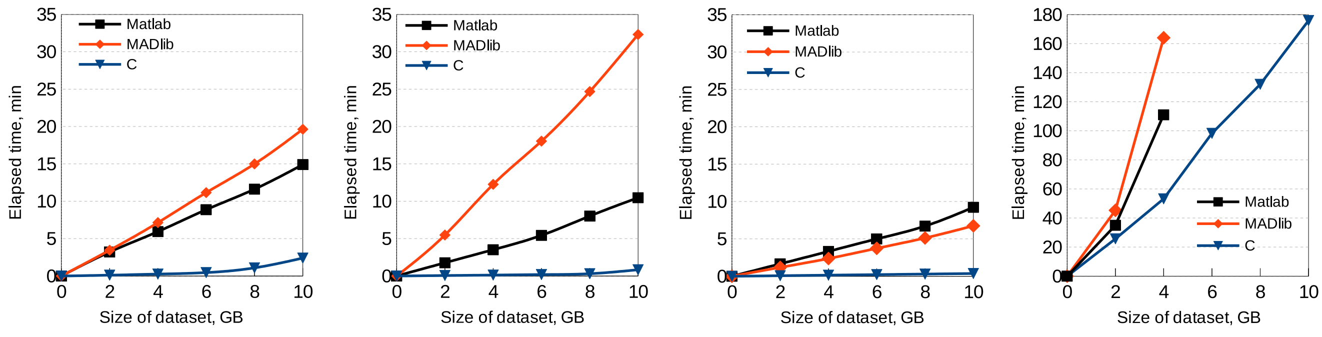
Next, we measure the time it takes each system to load data into main memory before executing the 3-line algorithm (we saw similar trends when testing other algorithms from the benchmark). In *cold-start*, we record the time to read the data from the underlying database or filesystem and run the algorithm. In *warm-start*, we first read the data into memory (e.g., into a Matlab array, or in PostgreSQL, we first run SELECT queries to extract the data we need) and then we run the algorithm. Thus, the difference between the cold-start and warm-start running times corresponds to the time it takes to load the data into memory.

Figure 6 shows the results on the real data set. The left bars indicate cold-start running times, whereas the right bars represent warm-start running times and are divided into three parts: T_1 is the time to compute the 10th and 90th quantiles, T_2 is the time to compute the regression lines and T_3 is the time to adjust the lines in case of any discontinuities in the piecewise regression model. Cold-start times are higher for all platforms, but Matlab and MADLib spend the most time loading data into their respective data structures, followed by System C. Overall, System C is easily the fastest and the most efficient at data loading—most likely due to efficient memory-mapped I/O. Also note that for each system, T_2, i.e., the time to run least-squares linear regression, is the most costly component of the 3-line algorithm.

Figure 6 suggests that System C is noticeably more efficient than Matlab even in the case of warm start, when Matlab has all the data it needs in memory. There are at least two possible explanations for this: Matlab's data structures are not as efficient as System C's, especially at the data sizes we are dealing with, or Matlab's implementation of linear regression and other statistical operators is not as efficient as our hand-crafted implementations within System C. We suspect it is the former. To confirm this hypothesis, we measured the running time of multiplying two randomly-generated 4000x4000 floating-point matrices in Matlab and System C. Indeed, Matlab took under a second, while System C took over 5 seconds.

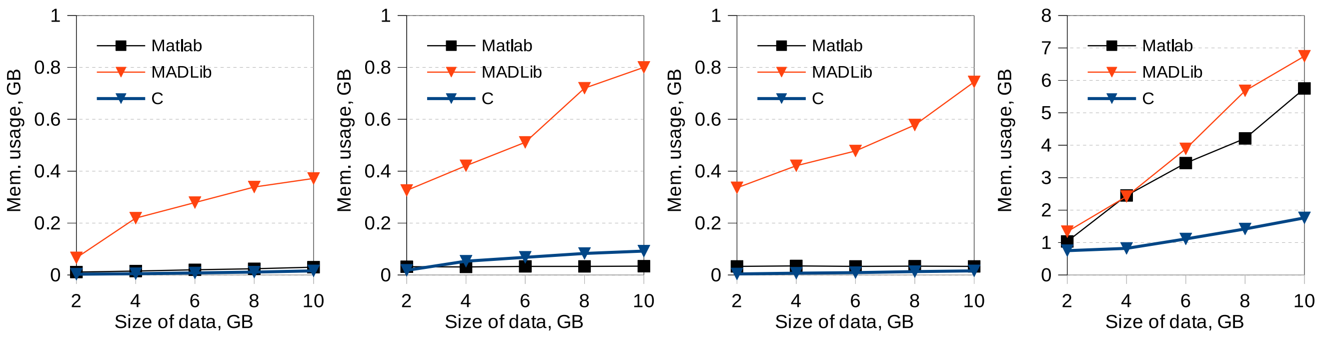
5.3.3 *Single-Threaded Results*

We now measure the cold-start running times of each algorithm in single-threaded mode (i.e., no parallelism). System C has configuration parameters that govern the level of parallelism, while for Matlab, we start a single instance, and for MADLib, we establish a single database connection. We use subsets of our real data sets with sizes between 2 and 10 GB for this experiment. The running time results are shown in Figure 7 for 3-line, PAR, his-



(a) 3-line (b) PAR (c) Histogram (d) Similarity

Figure 7: Single-threaded execution times of each algorithm using each system.



(a) 3-line (b) PAR (c) Histogram (d) Similarity

Figure 8: Memory consumption of each algorithm using each system.

Table 1

householdID	temperature	reading
int	*double*	*double*
1000	-4	0.35
1000	-3	0.26
...		
2000	-2	2.0
2000	3	1.1
...		

Table 2

householdID	temperature	reading
int	*double[]*	*double[]*
1000	[-4,-3,...]	[0.35,0.26,...]
...		
2000	[-2,3,...]	[2.0,1.1,...]
...		

Figure 9: Two table layouts for storing smart meter data in PostgreSQL.

togram construction and similarity search, from left to right. Note that the Y-axis of the rightmost plot is different: similarity search is slower than the other three tasks, and the Matlab and MADLib curves end at 4GB because the running time on larger data sets was prohibitively high. System C is the clear winner: it is a commercial system that is fast at data loading thanks to memory-mapped I/O, and fast at query execution since we implemented the required statistical operators in a low-level language. Matlab is the runner-up in most cases except for histogram construction, which is simpler than the other tasks and can be done efficiently in a database system without optimized vector and matrix operations. MADLib has the worst performance for 3-line, PAR and similarity search.

Figure 8 shows the corresponding memory consumption of each algorithm for each platform; the plots correspond to running the "free -m" command every five seconds throughout the runtime of the algorithms and taking the average. Matlab and System C have the lowest memory consumption; recall that for Matlab, we use separate files for different consumers' data and therefore the number of files that need to be in memory at any given time is limited.

In terms of the tested algorithms, 3-line has the lowest memory usage since it only requires the 10th and 90th percentile data points to compute the regression lines, not the whole time series. The memory footprint of PAR and histogram construction is higher because they both require the whole time series. The memory usage of similarity search is higher still, especially for Matlab and MADLib, both of which keep all the data in memory for this task. On the other hand, since System C employs memory-mapped files, it only loads what is required.

The relatively poor performance of MADLib may be related to its internal storage format. In the next experiment, we check if using the PostgreSQL *array* data type improves performance. Table 1 in Figure 9 shows the conventional row-oriented schema for smart meter data which we have used in all the experiments so far, with a household ID, the outdoor temperature, and the electricity consumption reading (plus the timestamp, which is not shown). That is, each data point of time time series is stored as a separate row, and a B-tree index is built on the household ID to speed up the extraction of all the data for a given consumer. Table 2 in Figure 9 stores one row for each consumer (household) and uses arrays to store all the temperature and consumption readings for the given consumer using the same positional encoding. Using arrays, the running time of 3-line on the whole 10 GB data set went down from 19.6 minutes to 11.3 minutes, which is faster than Matlab and Spark but still much slower than System C (recall the leftmost plot in Figure 7). The other algorithms also ran slightly faster but not nearly as fast as in System C: the PAR running time went down from 34.9 to 30 minutes, the histogram running time went down from 7.8 to 6.8 minutes, and the running time of similarity search (using 6400

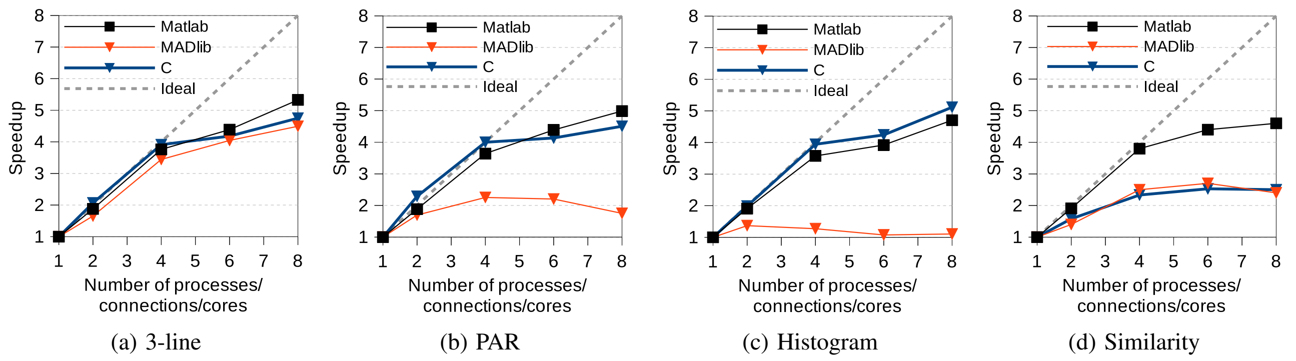


Figure 10: Speedup of execution time on a single multi-core server using the 10GB real dataset.

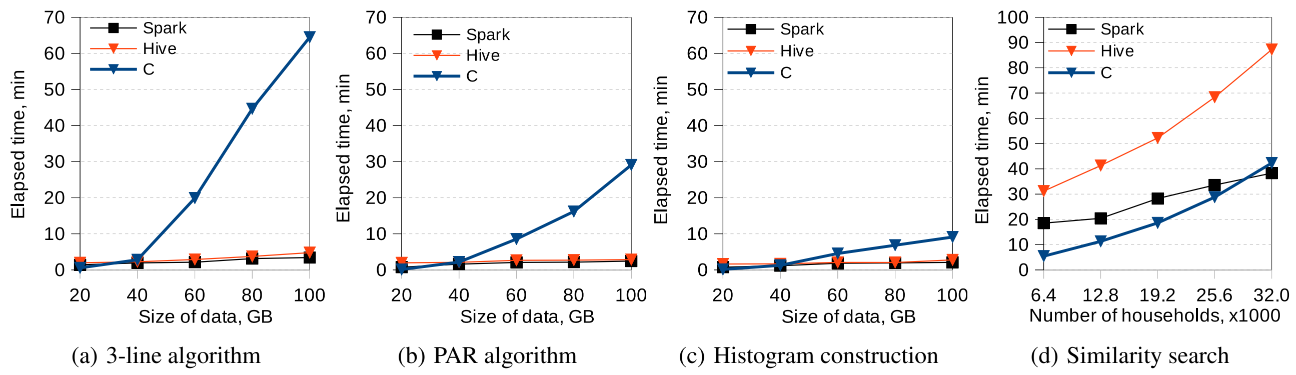


Figure 11: Execution times using large synthetic data sets.

households, which works out to about 2 GB) went down from 58.3 to 40.5 minutes. Finally, we also experimented with a table layout in between those in Table 1 and Table 2, namely one row per consumer per day, which resulted in running times in between those obtained from Table 1 and Table 2.

5.3.4 *Multi-Threaded Results*

We now evaluate the ability of the tested platforms to take advantage of parallelism. Our server has 4 cores with two hyper-threads per core and so we vary the number of processes from 1 to 8. Again, we can do so directly in System C, but we need to manually run multiple instances of Matlab and start up multiple database connections in MADLib. The histogram, 3-line and PAR algorithms are easy to parallelize as each thread can run on a subset of the consumers without communicating with the other threads. Similarity search is harder to parallelize because for each time series, we need to compute the cosine similarity to every other time series. We do this by running parallel tasks in which each task is allocated a fraction of the time series and computes the similarity of its time series with every other time series.

Figures 10(a)–10(d) show the speedup obtained by increasing the number of threads from 1 to 8 for each algorithm. Again, we continue to use the 10-GB real data set. Each plot includes a diagonal line indicating ideal speedup (i.e., using two connections or cores would be twice as fast as using one).

The results show that Matlab and System C can obtain nearly-linear speedup when the degree of parallelism is no greater than four. This makes sense since our server has four physical cores, and increasing the level of parallelism beyond four brings diminishing returns due to increasing resource contention (e.g., for floating point units) among hyper-threads. Matlab appears to scale better than MADLib, but this may be an artifact of how we simulate parallelism for these two platforms: Matlab instances effectively run in a shared-nothing fashion because each consumer's data are in a separate file, while MADLib uses multiple connections to the same database server, with each connection reading data from a single table.

5.4 Cluster Results

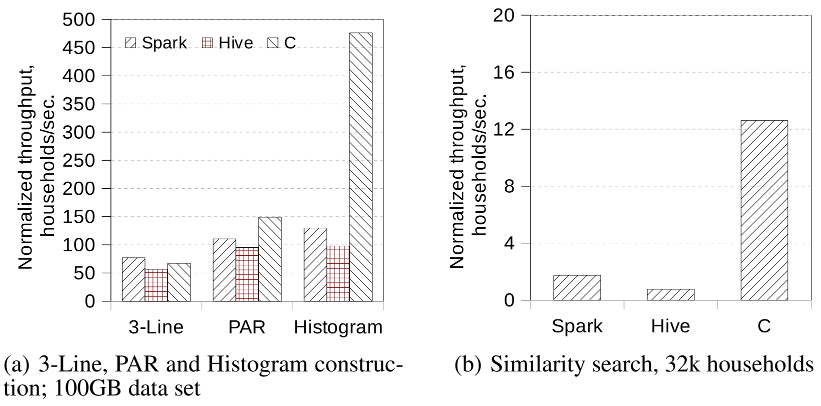
We now focus on the performance of Spark and Hive on a cluster using large synthetic data sets. We set the number of parallel executors for Spark and the number of MapReduce tasks for Hive to be up to 12 per node, which is the number of physical cores[8].

5.4.1 *System C vs. Spark and Hive*

In the previous batch of experiments, System C was the clear performance winner in a single-server scenario. We now compare System C against the two distributed platforms, Spark and Hive, on large synthetic data sets of up to 100GB (for similarity search, we use 6,000 up to 32,000 time series). This experiment is unfair in the sense that we run System C on the server (with maximum parallelism level of eight hyper-threads) but we run Spark and Hive on the cluster. Nevertheless, the results are interesting.

Figure 11 shows the running time of each algorithm. Up to 40GB data size, System C is keeping up with Spark and Hive despite running on a single server. Similarity search performance of System C

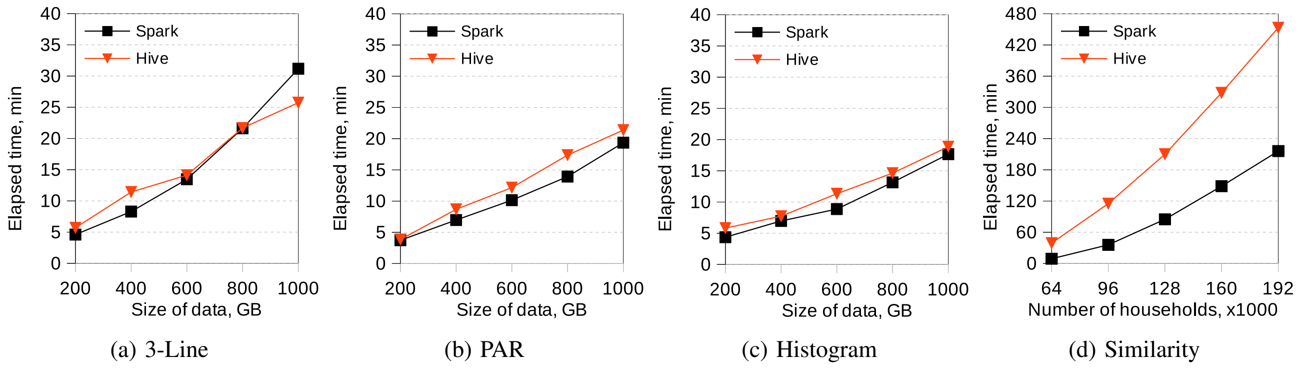
[8]We experimented with different values of these parameters and found that Spark was not sensitive to the number of parallel executors while Hive generally performed better with more MapReduce tasks up to a certain point.



(a) 3-Line, PAR and Histogram construction; 100GB data set

(b) Similarity search, 32k households

Figure 12: A comparison of throughput per server of System C, Spark and Hive.



(a) 3-Line

(b) PAR

(c) Histogram

(d) Similarity

Figure 13: Execution times using the first data format in Spark and Hive.

is also very good.

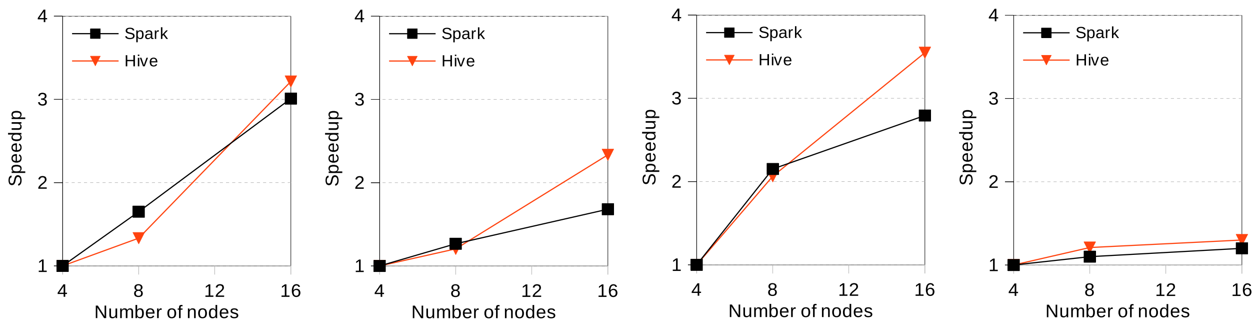
Figure 12 illustrates another way of comparing the three systems that is more fair. Part (a) shows the throughput, for 3-Line, PAR and histogram construction, in terms of how many households can be handled per second *per server* when using the 100GB synthetic data set. That is, we divide the total throughput of Spark and Hive by 16, the number of worker nodes in the cluster. Using this metric, even at 100GB, System C is competitive with Spark and Hive on 3-Line and PAR, and better on the simple algorithm of histogram construction. Similarly, part (b) shows that the throughput per server for similarity search is higher for System C at 32k households.

5.4.2 Spark vs. Hive using Different Data Formats

In this experiment, we take a closer look at the relative performance of Spark and Hive and the impact of the file format, using synthetic data sets up to a Terabyte. We use the default HDFS text file format, with default serialization, and without compression. The three options we test are: 1) one file (that may be partitioned arbitrarily) with one smart meter reading per line, 2) one file with one household per line (i.e., all the readings from a single household on a single line), and 3) many files, with one or more households per file (but no household scattered among many files), and one smart meter reading per line. Note that while the first format is the most flexible in terms of storage, it may require a *reduce* step for the tested algorithms since we cannot guarantee that all the data for a given household will be on the same server. The second and third options do not require a reduce step.

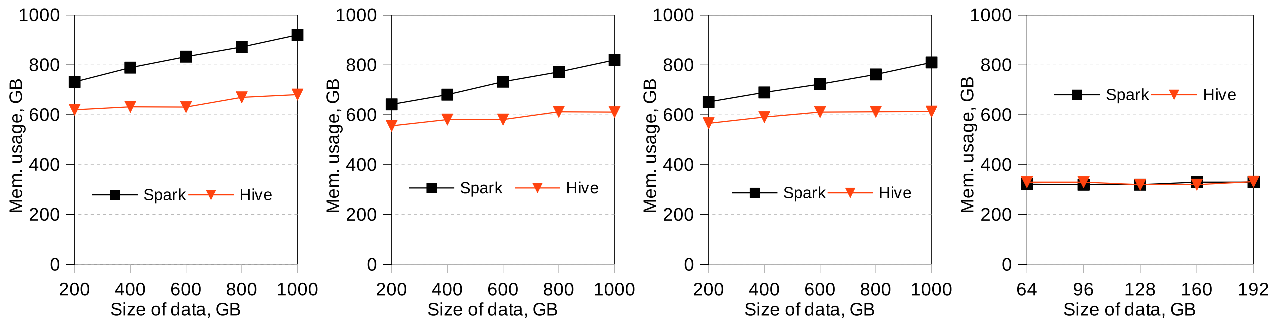
In Hive, we use three types of user-defined functions with the three file formats: generic UDF (user defined function), UDAF (user defined aggregation function) and UDTF (user defined table function). UDF and UDTF typically run at the map side for the scalar operations on a row, while UDAF runs at the reduce side for an aggregation operations on many rows. We use a UDAF for the first format since we need to collate the numbers for each household to compute the tested algorithms. We use a generic UDF for the second format, for which map-only jobs suffice. We use a UDTF for the third format since UDTFs can process a single row and do the aggregation at the map side, which functions as a *combiner*. For the third format, we also need to customize the file input format, which takes a single file as an input split. We overwrite the `isSplitable()` method in the `TextInputFormat` class by returning a `false` value, which ensures that any given time series is processed in a self-contained manner by a single mapper.

First data format. Figure 13 shows the execution time of the four tested algorithms on various data set sizes up to a Terabyte. Spark is noticeably faster for similarity search (in Hive, we implemented this as a self-join, which resulted in a query plan that did not exploit map-side joins, whereas in Spark we directly implemented similarity search as a MapReduce job with broadcast variables and map-side joins), slightly faster for PAR and histogram construction, and slower for 3-Line construction as the data size grows. Figure 14 shows the speedup relative to using only 4 out of 16 worker nodes for the Terabyte data set, with the number of worker nodes on the X-axis. Hive appears to scale slightly better as we increase the number of nodes in the cluster. Finally, Figure 15 shows the memory usage as a function of the data set size, computed the same way as in Figure 8. Spark uses more memory than Hive, especially as the data size increases. As for the different algorithms, 3-Line is



(a) 3-Line (b) PAR (c) Histogram (d) Similarity, 64k households

Figure 14: Speedup obtained using the first data format in Spark and Hive.



(a) 3-Line (b) PAR (c) Histogram (d) Similarity

Figure 15: Memory consumption of each algorithm in Spark and Hive.

the most memory-intensive because it requires temperature data in addition to smart meter data.

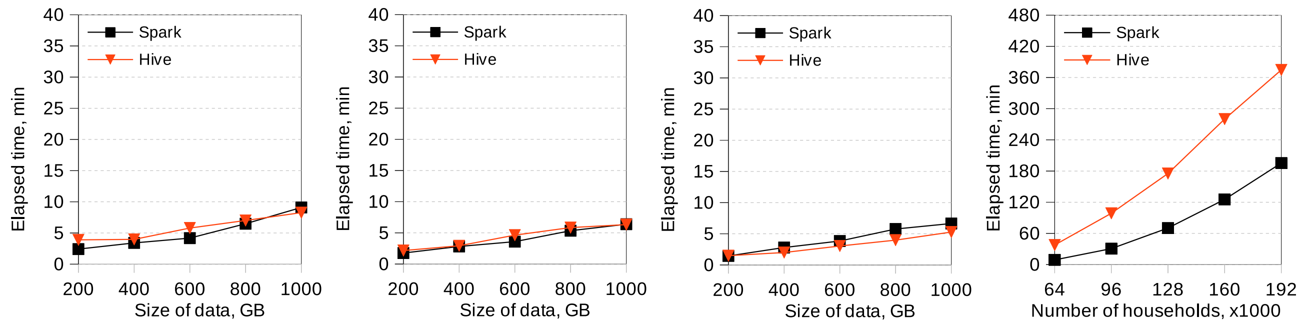
Second data format. Figure 16 and 17 show the execution times and the speedup, respectively, with one time series per line. For 3-Line, PAR and histogram construction, we do not require a reduce step. Therefore the running times are lower than for the first data format, in which a single time series may be scattered among nodes in the cluster. Spark and Hive are very close in terms of running time because they perform the same HDFS I/O. We also see a higher speedup than with the first data format thanks to map-only jobs, which avoid an I/O-intensive data shuffle among servers compared to jobs that include both map and reduce phases. Similarity search is slightly faster than with the first data format; most of the time is spent on computing the pair-wise similarities, and the only time savings in the second data format are due to not having to group together the readings from the same households. Note that similarity search still requires a reduce step to sort the similarity scores for each households and find the top-k most similar consumers.

Third data format. Here, we only use the 100GB data set with a total of 260,000 households and we vary the number of files from 10 to 10,000; recall that in the third data format, the readings from a given time series are guaranteed to be in the same file. We test two options in Hive: a UDTF with the customized file input format described earlier, and a UDAF in which a reduce step is required. We do not test similarity search since the distance calculations between pairs of time series cannot be done in one UDTF operation. Figure 18 and Figure 19 show the execution times and the speedup, respectively. Hive with UDTF wins in this format since it does not have to perform a reduce step. Furthermore, while Hive does not seem to be affected by the number of files, at least between 10 and 10,000, Spark's performance deteriorates as the number of files increases. In fact, we also experimented with more files, up to 100,000, and found that Spark was not even runnable due to "too many files open" exceptions.

5.5 Lessons Learned

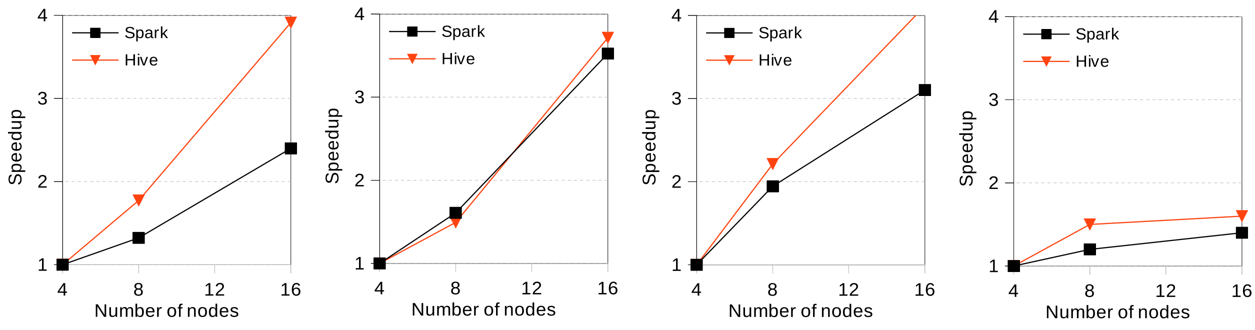
Our main finding is that System C, which is a commercial main-memory column store, is the best choice for smart meter analytics in terms of performance, provided that the resources of a single machine are sufficient. However, System C lacks a built-in machine learning toolkit and therefore we had to invest significant programming effort to build efficient analytics applications on top of it. On the other hand, Matlab and MADLib are likely to be more programmer-friendly but slower. Furthermore, we found that Matlab works better if each customer's time series is stored in a separate file and that PostgreSQL/MADLib works well when the smart meter data are stored using a hybrid row/column oriented format.

As for the two distributed solutions, Spark was slightly faster but Hive scaled slightly better as we increased the number of worker nodes. Moreover, we found Hive easier to use due to its DBMS-like features and a declarative language. Furthermore, we showed that the choice of data format matters; we obtained best performance when each time series was on a separate line, which eliminated the need to group data explicitly by household ID and thus avoided an I/O-intensive data shuffle among servers. This feature allows our implementations to remain competitive in terms of efficiency with respect to System C for 3-line and PAR, whereas cluster computing frameworks in general are known to suffer from poor efficiency compared to centralized systems [6].



(a) 3-Line (b) PAR (c) Histogram (d) Similarity

Figure 16: Execution times using the second data format in Spark and Hive.



(a) 3-Line (b) PAR (c) Histogram (d) Similarity, 64k households

Figure 17: Speedup obtained using the second data format in Spark and Hive.

6. CONCLUSION AND FUTURE WORK

Smart meter data analytics is an important new area of research and practice. In this paper, we studied smart meter analytics from a software performance perspective. We proposed a performance benchmark for smart meter analytics consisting of four common tasks, and presented a data generator for creating very large smart meter data sets. We implemented the proposed benchmark using five state-of-the-art data processing platforms and found that a main-memory column-store system offers the best performance on a single machine, but systems such as MADLib/PostgreSQL and Matlab are more programmer-friendly due to built-in statistical and machine learning operators. In cluster environments, we found Hive easier to use than Spark and not much slower. Compared to centralized solutions, we found Hive and Spark competitive in terms of efficiency for CPU-intensive data-parallel workloads (3-line and PAR).

We are currently building a smart meter analytics system that includes the four algorithms from the proposed benchmark and many more. As part of this ongoing project, we are investigating new ways of improving the efficiency and effectiveness of smart meter data mining algorithms, including parallel implementation. Another interesting direction for future work is to investigate real-time applications using high-frequency smart meters (which are not yet widely available, but are likely to become cheaper and more common in the future), such as alerts due to unusual consumption readings, using data stream processing technologies. Finally, we are interested in developing a general time series analytics benchmark for a wider range of applications.

7. ACKNOWLEDGEMENTS

We are grateful to Matt Cheah for developing a preliminary implementation of the 3-line algorithm using Apache Spark and evaluating its performance on a multi-core server.

8. REFERENCES

[1] J. M. Abreu, F. P. Camara, and P. Ferrao, Using pattern recognition to identify habitual behavior in residential electricity consumption, Energy and Buildings, 49:479-487, 2012.

[2] G. Acs and C. Castelluccia, I have a DREAM (DiffeRentially privatE smArt Metering), in Conf. on Information Hiding, 118-132, 2011.

[3] A. Albert, T. Gebru, J. Ku, J. Kwac, J. Leskovec, and R. Rajagopal, Drivers of variability in energy consumption, in ECML-PKDD DARE Workshop on Energy Analytics, 2013.

[4] A. Albert and R. Rajagopal, Building dynamic thermal profiles of energy consumption for individuals and neighborhoods, in IEEE Big Data Conf., 723-728, 2013.

[5] A. Albert and R. Rajagopal, Smart meter driven segmentation: what your consumption says about you. IEEE Transactions on Power Systems, 4(28), 2013.

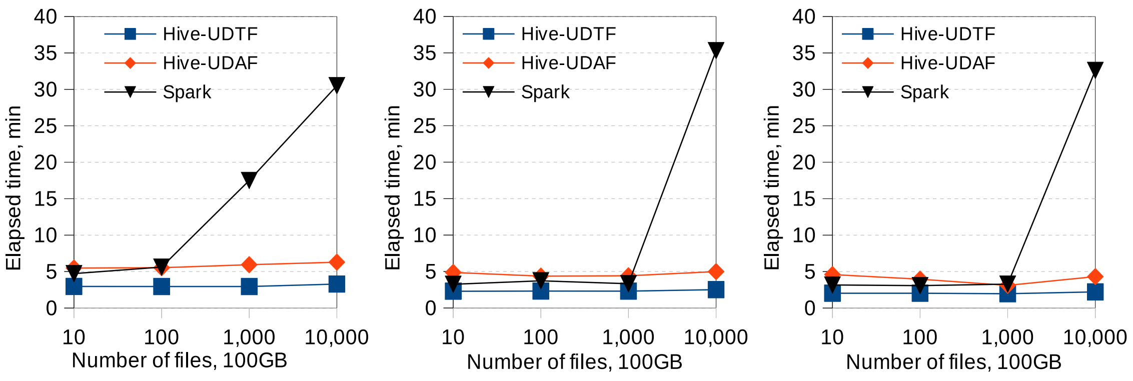
[6] E. Anderson and J. Tucek, Efficiency Matters!, SIGOPS Operating Systems Review, 44(1):40-45, 2010.

[7] C. Anil, Benchmarking of Data Mining Techniques as Applied to Power System Analysis, Master's Thesis, Uppsala University, 2013.

[8] O. Ardakanian, N. Koochakzadeh, R. P. Singh, L. Golab, and S.Keshav, Computing Electricity Consumption Profiles from Household Smart Meter Data, in EnDM Workshop on Energy Data Management, 140-147, 2014.

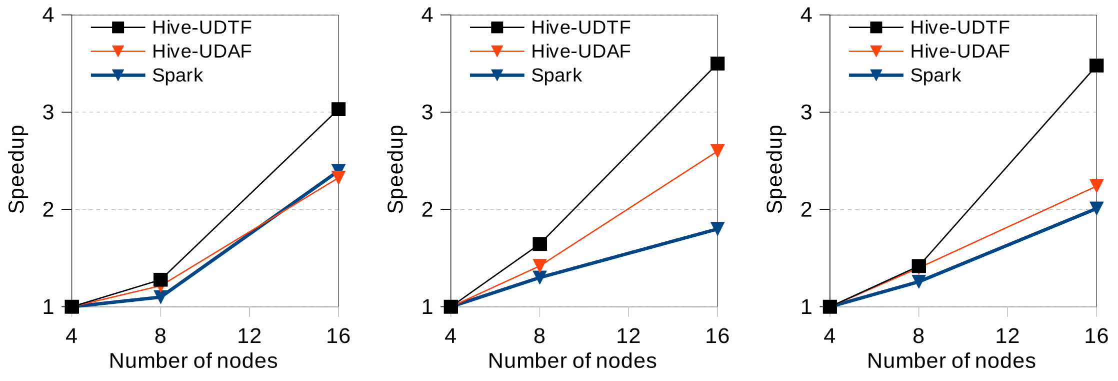
[9] M. Arlitt, M. Marwah, G. Bellala, A. Shah, J. Healey, B. Vandiver, IoTA bench: an Internet of Things Analytics benchmark, HP Laboratories Technical Report, HPL-2014-75.

[10] B. J. Birt, G. R. Newsham, I. Beausoleil-Morrison, M. M. Armstrong, N. Saldanha, and I. H. Rowlands, Disaggregating



(a) 3-Line (b) PAR (c) Histogram

Figure 18: Execution times using the third data format in Spark and Hive.



(a) 3-Line (b) PAR (c) Histogram

Figure 19: Speedup obtained using the third data format in Spark and Hive, 100 files, 1GB per file.

Categories of Electrical Energy End-use from Whole-house Hourly Data, Energy and Buildings, 50:93-102, 2012.

[11] N. Bruno and S. Chaudhuri, Flexible database generators, in VLDB, 1097-1107, 2005

[12] G. Chicco, R. Napoli, and F. Piglione, Comparisons among Clustering Techniques for Electricity Customer Classification, IEEE Trans. on Power Systems, 21(2):933-940, 2006.

[13] M. Espinoza, C. Joye, R. Belmans, and B. DeMoor, Short-term Load Forecasting, Profile Identification, and Customer Segmentation: A Methodology Based on Periodic Time Series, IEEE Trans. on Power Systems, 20(3):1622-1630, 2005.

[14] V. Figueiredo, F. Rodrigues, Z. Vale, and J. Gouveia, An electric energy consumer characterization framework based on data mining techniques, IEEE Trans. on Power Systems, 20(2):596-602, 2005.

[15] M. Ghofrani, M. Hassanzadeh, M. Etezadi-Amoli and M. Fadali, Smart Meter Based Short-Term Load Forecasting for Residential Customers, North American Power Symposium (NAPS), 2011.

[16] Greentech Media Research, The Soft Grid 2013-2020: Big Data & Utility Analytics for Smart Grid, http://www.greentechmedia.com/research/report/the-soft-grid-2013.

[17] J. M. Hellerstein, C. Re, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, and A. Kumar, The MADlib Analytics Library: or MAD Skills, the SQL, PVLDB, 5(12):1700-1711, 2012.

[18] R.-S. Jeng, C.-Y. Kuo, Y.-H. Ho, M.-F. Lee, L.-W. Tseng, C.-L. Fu, P.-F. Liang, L.-J. Chen, Missing Data Handling for Meter Data Management System, in e-Energy Conf., 275-276, 2013.

[19] E. Keogh, and S. Kasetty, On the Need for Time Series Data Mining Benchmarks: A Survey and Empirical Demonstration, Data Mining and Know. Disc. (DMKD), 7(4):349-371, 2003.

[20] Y. Liu, S. Hu, T. Rabl, W. Liu, H.-A. Jacobsen, K. Wu, J. Chen, J. Li, DGFIndex for Smart Grid: Enhancing Hive with a Cost-Effective Multidimensional Range Index. PVLDB 7(13): 1496-1507, 2014.

[21] F. Mattern, T. Staake, and M. Weiss, ICT for green - How computers can help us to conserve energy, in e-Energy Conf., 1-10, 2010.

[22] A. J. Nezhad, T. K. Wijaya, M. Vasirani, K. Aberer, SmartD: smart meter data analytics dashboard, in e-Energy Conf., 213-214, 2014.

[23] T. Rasanen, D. Voukantsis, H. Niska, K. Karatzas and M. Kolehmainen, Data-based method for creating electricity use load profiles using large amount of customer-specific hourly measured electricity use data, Applied Energy, 87(11):3538-3545, 2010.

[24] B. A. Smith, J. Wong, and R. Rajagopal, A Simple Way to Use Interval Data to Segment Residential Customers for Energy Efficiency and Demand Response Program Targeting, in ACEEE Summer Study on Energy Efficiency in Buildings, 2012.

[25] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, R. Murthy, Hive - A Warehousing Solution Over a Map-Reduce Framework. PVLDB 2(2): 1626-1629, 2009.

[26] G. Tsekouras, N. Hatziargyriou, and E. Dialynas, Two-stage Pattern Recognition of Load Curves for Classification of Electricity Customers, IEEE Trans. on Power Systems, 22(3):1120-1128, 2007.

[27] T. K. Wijaya, J. Eberle and K. Aberer, Symbolic representation of smart meter data, in EnDM Workshop on Energy Data Management, 242-248, 2013.

[28] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, Spark: Cluster Computing with Working Sets, in USENIX Conf., 10, 2010.

Crowd-Selection Query Processing in Crowdsourcing Databases: A Task-Driven Approach

Zhou Zhao[†][*], Furu Wei[§], Ming Zhou[§], Weikeng Chen[†], Wilfred Ng[†]
[†]Department of Computer Science and Engineering, Hong Kong University of Science and Technology
[§]Microsoft Research Asia, Beijing, China
[†]{zhaozhou, wilfred}@cse.ust.hk, [†]wchenad@connect.ust.hk,
[§]{fuwei,mingzhou}@microsoft.com

ABSTRACT

Crowd-selection is essential to crowdsourcing applications, since choosing the right workers with particular expertise to carry out specific crowdsourced tasks is extremely important. The central problem is simple but tricky: given a crowdsourced task, who is the right worker to ask? Currently, most existing work has mainly studied the problem of crowd-selection for simple crowdsourced tasks such as decision making and sentiment analysis. Their crowd-selection procedures are based on the trustworthiness of workers. However, for some complex tasks such as document review and question answering, selecting workers based on the latent category of tasks is a better solution.

In this paper, we formulate a new problem of task-driven crowd-selection for complex tasks. We first develop a bayesian generative model to exploit "who knows what" for the workers in the crowdsourcing environment. The model provides a principle and natural framework for capturing the latent skills of workers as well as the latent categories of crowdsourced tasks. The inference of the latent skills of workers is based on past resolved crowdsourced tasks with feedback scores. We assume that the feedback scores can illustrate the performance of the workers for the tasks. We then devise a variational algorithm that transforms the latent skill inference with the proposed model into a standard optimization problem, which can be solved efficiently. We verify the performance of our method through extensive experiments on the data collected from three well-known crowdsourcing platforms for question answering tasks such as Quora, Yahoo ! Answer and Stack Overflow.

1. INTRODUCTION

In recent years, crowdsourcing techniques [18] have attracted a lot of attention due to their effectiveness in real-life applications. They tackle the tasks including image tagging [19], decision making [12, 16] and natural language processing, which are hard for computers, but relatively easy for human workers. Some successful crowdsourcing examples for question answering tasks that appear include Quora [23], Yahoo ! Answer [5] and Stack Overflow [1], where users submit questions and get answers from the crowd. Using crowdsourcing techniques, these tasks can be solved well by human workers.

Despite the success of crowdsoucing techniques, the crowd-selection still remains challenging. Earlier approaches usually focus on the problem of crowd-selection for simple tasks where the selection procedure is based on the trustworthiness of workers [12, 7, 16, 31, 2]. However, in many cases, the task-driven crowd-selection is a better solution. Consider a question answering task t, "What are the advantages of B+ Tree over B Tree?". The existing crowd-selection procedure may not work in this case. Since the trustworthy workers may not be skilled in the area of computer science, therefore, they may not be able to answer this question.

In this paper, we formulate a new problem of task-driven crowd-selection. We focus on addressing the following three challenging issues.

- **Crowd Modeling.** We extend the crowd modeling from single-dimensional trustworthiness to multi-dimensional skills for task-driven crowd-selection. Thus, we model the strengths and weaknesses of workers on latent category of tasks. However, the existing latent skill models [28, 33] based on *Multinomial distribution*[1] are difficult to apply to our problem. The skill values on the latent categories are normalized to one for the property of Multinomial distribution. Thus, the skills of workers on specific latent categories cannot be comparable. For example, the latent skills of worker w^i is (CS 0.9, Math 0.1) and the latent skills of worker w^j is (CS 0.8, Math 0.2). Given a CS-based task above, the existing models select w^i since its skill value on CS is higher. However, it might be that w^j is better on CS while w^j solves more Math-based tasks. Thus, the skill value of their models cannot infer the strengths and weaknesses of workers on specific latent categories. We propose a novel crowd model to tackle this problem.

- **Latent Skill Inference.** The probabilistic inference for the latent skills of workers is based on the past resolved tasks. The existing inference approaches are either based on the content of tasks [28, 33, 32] or answer consistency with other workers [19]. However, we argue that the skills of workers are not necessarily related to the number of their resolve tasks or the difference between their answers and others. We consider the feedback score of the resolved tasks to illustrate the

[*]The work was done when the first author was visiting Microsoft Research, Beijing, China.

[1]http://en.wikipedia.org/wiki/Multinomial_distribution

10.5441/002/edbt.2015.35

latent skills of workers. We argue that the feedback score is a better quality measure for the job done by workers. Nowadays, the feedback score has been widely used in crowdsourcing platforms such as the satisfactory rate in Amazon Mechanical Turk [2] and Crowdflower [3] , and the thumbs-up in Question Answering System like Quora, Yahoo ! Answer and Stack Overflow. We propose a novel latent skill inference method based on resolved tasks with feedback scores.

- **Incremental Crowd-Selection.** The crowdsourced tasks are always in quantity and arriving in high speed. Therefore, it is time consuming for latent category inference of the crowdsourced tasks in batch. In this work, we first devise a bayesian model that builds a latent category space and infers the skills of workers on that space based on the past resolved tasks. Next, we propose an incremental latent category inference algorithm that projects the newly coming tasks into the existing latent category space. Then, we can select the workers who are skilled in these categories to solve the tasks.

In this paper, we propose a bayesian model for task-driven crowd-selection. Our contributions are summarized as follows:

- We first propose the problem of task-driven crowd-selection for crowdsourced tasks.
- We propose a novel crowd model for modeling the skills on latent category space that enables the task-driven crowd-selection. The model also makes the skills of workers on specific latent task categories become comparable.
- We make the use of the feedback scores of the resolved tasks for latent skill inference. We consider the the feedback scores as the quality measure of the job done by the workers.
- We devise a variational algorithm that transforms the complex latent skill inference problem into a high-dimensional optimization problem, which can be solved efficiently.
- We develop an incremental crowd-selection algorithm that chooses the right workers for coming crowdsourced tasks in the real time.
- We evaluate our algorithm on the data collected from three well-known crowdsourcing applications Quora, Yahoo ! Answer, and Stack Overflow. For all datasets tested, we show that the quality of the selected workers based on our crowd model is more superior to that of other existing worker models including TSPM [8] and DRM [28].

Organizations. The rest of the paper is organized as follows. Section 2 gives an overview of the architecture of our method. Section 3 surveys the related works. Section 4 presents our bayesian model for task-driven crowd-selection. Section 5 introduces a variational algorithm for our proposed model. We present the incremental crowd-selection procedure in Section 6. We report the experimental results in Section 7 and conclude the paper in Section 8.

2. OVERVIEW

In this section, we give an overview of the architecture for task-driven crowd-selection.

We illustrate the architecture of our task-driven crowd-selection system in Figure 1. The core component of our system is crowd manager. The main functionalities of crowd manager are latent skill inference for workers and choose the right crowd for given crowdsourced tasks. The crowd model is stored in the crowd databases which support crowd insertion, crowd update and crowd retrieval. The red lines show the process of latent skill inference for workers as well as build latent category space for tasks, which is based on resolved tasks with feedback scores. The crowd databases are then updated. The blue line show the process of task-driven crowd-selection. Given a coming crowdsourced task, the crowd manager first projects it into the built latent category space. Next, the crowd manager returns the workers online as the candidate crowd for this task. The crowd manager then ranks the workers who are skilled in this task. The top ranked workers are chosen for solving the task. After that, the task dispatcher distributes this task to the selected workers. Finally, the system keeps collect the answers return by the selected workers.

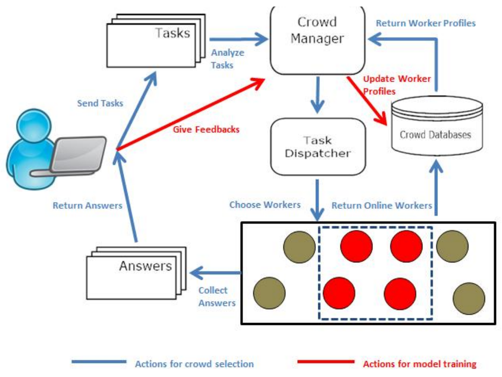


Figure 1: An Architecture for Task-Driven Crowd-Selection

In summary, our task-driven crowd-selection system can automatically ask the right crowd to process the crowdsourced tasks. The system is able to incrementally project the coming tasks to the existing latent category space such that the workers can be chosen in the real time. In the following sections, we present the idea and the methods of implementing this task-driven crowd-selection system.

3. RELATED WORK

Crowdsourcing has been widely used to solve challenging problems by human intelligence in comprehensive areas. Some successful applications that appear include CrowdDB [6], Qurk [14], CrowdSearch [29], HumanGS [15] and CDAS [12].

Recently, the crowdsourcing techniques have been applied in several research areas such as database management, machine learning and information retrieval. The crowdsourcing techniques on entity resolution were studied in [24, 26]. CrowdScreen [16] applied the crowdsourcing techniques in decision making. Guo et al. [9] and Venetis et al. [21] studied the problem of finding maximum element in the crowdsourcing databases. In [20], Trushkowsky et al proposed a method for crowdsourced enumerated query. In [4], Davidson et al proposed the top-k and group-by queries on crowdsourcing databases. Kaplan et al [11] aimed to select the right question for planing queries. Zhang et al [30] reduced the uncertainty for schema matching using crowdsourcing techniques. Marcus et al [13] studied the count query with the crowd. Park et al [17] aimed to find a best query query plan for crowdsourced data. Welinder et al [25] and Gao et al [7] proposed crowdsourcing based online algorithm to find the ground truth. Wu et al [27] studied the query

[2]https://www.mturk.com/

[3]http://crowdflower.com/

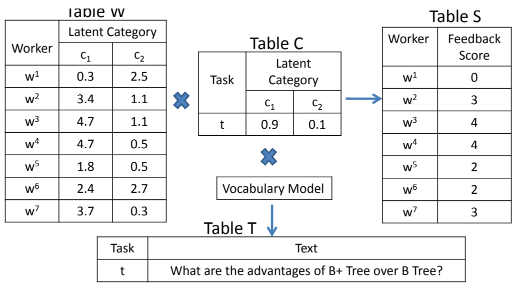


Table W

Worker	Latent Category	
	c_1	c_2
w^1	0.3	2.5
w^2	3.4	1.1
w^3	4.7	1.1
w^4	4.7	0.5
w^5	1.8	0.5
w^6	2.4	2.7
w^7	3.7	0.3

Table C

Task	Latent Category	
	c_1	c_2
t	0.9	0.1

Table S

Worker	Feedback Score
w^1	0
w^2	3
w^3	4
w^4	4
w^5	2
w^6	2
w^7	3

Table T

Task	Text
t	What are the advantages of B+ Tree over B Tree?

Figure 2: An Example of Generating Feedback Scores

processing over sensitive crowdsourced data.

The problem of crowd-selection is still the key component in these crowdsourcing applications, which has been studied in [12, 2, 3, 7]. However, these existing methods choose the crowd based on the trustworthiness of the workers, which may not be effective for complex crowdsourced tasks. In this paper, we study the problem of task-driven crowd-selection. We extend the crowd model from single-dimensional trustworthiness to multi-dimensional skills and propose a bayesian model for the skill variance of workers.

The most related works are TSPM [8] and DRM [28] for modeling the skills of workers. However, these methods model the skills of workers based on *Multinomial Distribution*, which we argue its limitation that it cannot distinguish the strengths and weaknesses of workers on specific latent category of the crowdsourced tasks in Section 1. In our experimental studies, we also show that the crowd-selection based on our novel crowd model outperforms both TSPM and DRM.

4. A BAYESIAN MODEL FOR TASK-DRIVEN CROWD-SELECTION

In this section, we present a bayesian model to infer the latent skills for workers as well as build the latent category of tasks for task-driven crowd-selection. The model exploits "who knows what" based on resolved tasks with feedback scores. Specifically, we calculate the posterior distribution of all possible worker skills, and find the most probable one with the maximum probability. Intuitively, this model best explains the feedback scores for the resolved tasks.

We start by illustrating an example of generating feedback scores on the jobs for a crowdsourced task, illustrated in Figure 2. After that, we introduce some basic notions and notations in Section 4.1, and define the problem in Section 4.2. Then, we present a generative process for task-driven crowd-selection in Section 4.3 and define the bayesian model in Section 4.4. We give a summary of notions and notations in Table 1.

Consider a question answering task "What are the advantages of B+ Tree over B Tree?" from Quora in Figure 2. We give the latent categories of this task in Table C. We assume that the vocabularies of this task are generated by its latent categories and the language model. The language model is vocabulary distribution over latent task categories. We can see that seven workers answered this question task and returned their answers. The latent skills of the workers are given in Table W and the feedback scores for their answers are given in Table S. The feedback scores are the number of thumbs-up for their returned answers in Quora. We assume that the feedback scores for the workers are proportional to their skills (i.e. $S \approx WC^T$). For instance, the feedback score of worker w_3 can be interpreted as $w_1^3 \times c_1 + w_2^3 \times c_2 = 4.7 \times 0.9 + 1.1 \times 0.1 \approx 4$.

Table 1: Summary of Notations

Notation	Meaning
T	A collection of N tasks $\{t^1, \ldots, t^N\}$
W	Latent skills of M workers $\{w^1, \ldots, w^M\}$
C	Latent categories on N tasks $\{c^1, \ldots, c^N\}$
A	Task assignment $\{a_{11}, \ldots, a_{ij}, \ldots, a_{MN}\}$
S	Feedback scores $\{s_{11}, \ldots, s_{ij}, \ldots, s_{MN}\}$
L	Number of vocabularies in the task
v_p^j	The p-th vocabulary in task t_j
z_p^j	Latent category of vocabulary v_p^j in task t_j
w^i	Latent skills of a worker $w^i = \{w_1^i, \ldots, w_K^i\}$
c^j	Latent categories of a task $c^j = \{c_1^j, \ldots, c_K^j\}$
Σ_c,ν_c	Prior for latent category
Σ_w,ν_w	Prior for worker skill
$\beta_{z_p^j}$	Prior for vocabulary

4.1 Notation

In this section, we introduce the notation used in the paper.

4.1.1 *Crowdsourced Task T*

The crowdsourced task t_j is represented as a bag of vocabularies $t_j = \{(v_1, \#v_1)$, $(v_2, \#v_2)$, …, $(v_L, \#v_L)\}$ where $\#v_p$ is the number of vocabulary v_p in task t_j and L is the number of vocabularies. For example, the task t_j in Figure 2 can be represented as $t_j = \{(advantage, 1)$, $(B, 1)$, $(B+, 1)$, $(over, 1)$, $(tree, 2)$, $(what, 1)\}$. We denote a collection of N crowdsourced tasks as T.

4.1.2 *Latent Task Category C*

The latent category of a task t_j is considered as a probability distribution $c^j = \{c_1^j, c_2^j, \ldots, c_K^j\}$ where K is the number of latent categories. We consider that c_k^j is the probability of task t_j in category c_k. The sum of the probability of a task t_j on the latent categories is $c_1^j + c_2^j \ldots + c_K^j = 1$. For example, the latent category of task t_j in Figure 2 is $c^j = \{c_1^j, c_2^j\} = \{0.9, 0.1\}$. We consider a collection of latent category of N crowdsourced tasks as C.

4.1.3 *Latent Worker Skill W*

The latent skills of worker w^i on K categories is $w^i = \{w_1^i$, w_2^i, …, $w_K^i\}$ where w_k^i is a positive real number illustrating the ability of the worker on the latent category c_k. For example, the latent skills of workers w^2 is $\{w_1^2, w_2^2\} = \{3.4, 1.1\}$ and the latent skills of worker w^1 is $\{w_1^1, w_2^1\} = \{0.3, 2.5\}$. For latent category c_1 based task, employing w^2 is a better choice. On the other hand, the performance of w^2 is better on c_2 based task. We denote a collection of the latent skills of M workers as W. We explain the inference of W in Section 4.3.

4.1.4 *Task Assignment A*

The task assignment A is a $N \times M$ binary matrix where the entry a_{ij} indicates the assignment of task t_j to worker w^i (i.e. a_{ij} can be either 0 or 1). In this paper, we assume that a task can be assigned to more than one worker and a worker can have multiple tasks. For example, the assignment of task t_j and worker w^1 is $a_{1j} = 1$ in Figure 2. The task assignment A can be incremented when new tasks are resolved or new workers are involved in the crowdsourcing environment.

4.1.5 *Task Feedback Score S*

The task feedback score S is a $N \times M$ matrix where the entry s_{ij} indicates the score of the job done by worker w^i on task t_j. For example, the feedback score of worker w^2 on task t_j is 3 in Figure 2. The range of feedback score depends on the specific crowdsourcing applications. Here, we introduce two types of feedback scores used in this work.

- **Best Answer.** We consider the best answer as the feedback to illustrate the quality of answers in Yahoo ! Answer. The best answer is given by the question asker. Consider a resolved question answering task t_j. For the worker w^i receives best answer, the feedback score on the work is $s_{ij} = 1$. For other workers, we define feedback scores for them based on Jaccard distance between their answers and the best answer where $0 \leq s_{kj} \leq 1$ and $k \neq i$.

- **Thumbs-up.** We regard thumbs-up as the feedback to illustrate the quality of answers in both Quora and Stack Overflow. The thumbs-up for answers are given by the users of Quora and Stack Overflow. We consider the number of thumbs-up as the feedback score s_{ij}.

4.2 Problem Definition

We now formulate the problem of task-driven crowd-selection as follows:

Given a task t_j, how to choose the right online workers who are skilled in solving the task? We build a bayesian model based on resolved crowdsourced task (T, A, S) such that the following computational issues are effectively addressed: (1) For the coming task t_j, it can be projected to the latent category space c^j of our model. (2) After solving the task, the skills of workers involved can be updated. Thus, the task-driven crowd-selection is based on the strengths of workers on the latent category of the task $w^i(c^j)^T$. The top-k crowd-selection is to find a subset R of k workers such that

$$R = \arg\max_{|R|=k} \sum_{i \in R} w^i (c^j)^T \tag{1}$$

where $(c^j)^T$ is a transposed vector of latent category c^j.

4.3 A Generative Process

In this section, we illustrate the generative process for the feedback scores on the task-driven crowd-selection. For brevity, we show the generative process of the feedback scores $S = \{s_{11}, \ldots, s_{NM}\}$ based on N crowdsourced tasks $T = \{t_1, t_2, \ldots, t_N\}$ and M workers $W = \{w^1, w^2, \ldots, w^M\}$.

Now, we denote a set of model parameters $\varphi = \{W, \Sigma_w, C, \Sigma_c, \tau, \beta_c\}$. The parameters W and Σ_w are the prior for latent worker skill while C and Σ_c are the prior for latent task category. The parameter τ is variance between the feedback score and the predictive performance of the workers. The parameter β_c is the language model that generates the vocabularies of the task based on latent category c. We outline the generative process in Algorithm 1.

4.3.1 *Generating Latent Worker Skill W*

For the latent skills of worker $w^i \in W$, we assume that the skills on the latent categories are generated from a Normal distribution, given by

$$\begin{aligned} w^i &\sim Normal(\mu_w, \Sigma_w) \\ &\sim \frac{1}{(2\pi)^{K/2}|\Sigma_w|^{1/2}} \exp\{-\frac{1}{2}(w^i - \mu_w)^T \Sigma_w^{-1} (w^i - \mu_w)\} \end{aligned} \tag{2}$$

where μ_w is the mean of the skills on latent categories and Σ_w is the correlation of skills on latent categories. It is a generalized way to model the skills on latent categories while a special way is to assume the independence of skills on latent categories. In that case, Σ_w is a diagonal matrix.

4.3.2 *Generating Latent Task Category C*

For the latent category of task $t_j \in T$, we assume that c^j is generated from a Normal distribution, given by

$$\begin{aligned} c^j &\sim Normal(\mu_c, \Sigma_c) \\ &\sim \frac{1}{(2\pi)^{K/2}|\Sigma_c|^{1/2}} \exp\{-\frac{1}{2}(c^j - \mu_c)^T \Sigma_c^{-1} (c^j - \mu_c)\} \end{aligned} \tag{3}$$

where μ_c is the latent category distribution for all the crowdsourced tasks. The parameter Σ_c is the correlation of latent category distribution for all the crowdsourced tasks.

4.3.3 *Generating Task Vocabulary V*

For the latent category of vocabulary v_p^j in task t_j, we assume z_p^j is generated from a discrete distribution, given by

$$\begin{aligned} z_p^j &\sim Discrete(logistic(c^j)) \\ &\sim \frac{\exp(c_k^j)}{\sum_{k=1}^K \exp(c_k^j)}. \end{aligned} \tag{4}$$

where $logistic(c^j)$ is a logistic function that transforms the latent category c^j to a discrete distribution.

Based on the latent category z_p^j, the vocabulary is generated from a discrete distribution, given by

$$v_p^j \sim \beta_{z_p^j} = p(v_p^j | \beta, z_p^j) \tag{5}$$

where β is a vocabulary distribution over latent categories. The parameter β is used as the language model to generate the vocabularies of all the crowdsourced tasks.

4.3.4 *Generating Task Feedback Score S*

For the feedback score s_{ij} on the task t_j assigned to the worker w^i, we assume that s_{ij} is generated from a Normal distribution, given by

$$\begin{aligned} s_{ij} &\sim Normal(w^i(c^j)^T, \tau) \\ &\sim \frac{1}{\tau\sqrt{2\pi}} \exp\{-\frac{(s_{ij} - w^i(c^j)^T)^2}{2\tau^2}\} \end{aligned} \tag{6}$$

where the parameter τ is the variance of the Normal distribution. The product $w^i(c^j)^T$ is the predictive performance of worker w^i on the task t^j.

We now present the details of the generative process for the feedback scores S on the crowdsourced task T in Algorithm 1. Algorithm 1 generates the skills of workers by Normal distribution with parameters μ_w and Σ_w from Line 1 to Line 3. Next, Algorithm 1 generates latent categories for the crowdsourced tasks by Normal distribution with parameters μ_c and Σ_c in Line 5. The latent category of the vocabulary z_p^j is generated by a discrete distribution with logistic function based on c^j in Line 7. Then, the vocabularies of task t_j is generated by the language model β and the latent category for vocabularies z^j in Line 8. After that, Algorithm 1 generates the feedback scores S on the workers for the tasks T from Line 1 to Line 15. The feedback score is generated by Normal distribution based on the predictive performance of workers on the task $w^i(c^j)^T$. Finally, Algorithm 1 returns the feedback score S in Line 16.

Algorithm 1 Generating feedback scores for resolved tasks

Input: A set of tasks T, task assignment A, a set of model parameters $\varphi = \{W, \Sigma_w, C, \Sigma_c, \tau, \beta_c\}$
Output: Feedback scores S

```
 1: for each worker $w^i \in W$ do
 2:     Choose skills $w^i$ by Equation 2.
 3: end for
 4: for each task $t_j \in T$ do
 5:     Choose latent category $c^j$ by Equation 3
 6:     for each vocabulary $v_p^j \in t_j$ do
 7:         (a) Choose a latent category $z_p^j$ by Equation 4
 8:         (b) Choose the text for vocabulary $v_p^j$ by Equation 5
 9:     end for
10: end for
11: for each crowdsourced task $t^j \in T$ do
12:     for each employed worker $w^i \in A_j$ do
13:         Generate feedback score $s_{ij}$ by Equation 6
14:     end for
15: end for
16: return feedback scores $S$
```

4.4 Model Definition

In the previous discussion, we described a generative process for the feedback scores on the crowdsourced tasks. We now formally define a bayesian model that represents the underlying joint distribution over the vocabularies of tasks V and the feedback scores of tasks S.

Given a set of model parameters $\varphi = \{\mu_w, \Sigma_w, \mu_c, \Sigma_c, \tau, \beta\}$, and task assignment A, we factorize the joint distribution over V and S, given by

$$\begin{aligned} p(V, S|A, \varphi) &= \int_C \int_W p(W|\mu_w, \Sigma_w) p(C|\mu_c, \Sigma_c) p(Z|C) \\ &\times \; p(V|Z, \beta) p(S|WC^T, \tau) dC dW \end{aligned}$$

where

$$\begin{aligned} p(W|\mu_w, \Sigma_w) &= \prod_{w^i \in W} p(w^i|\mu_w, \Sigma_w), \\ p(C|\mu_c, \Sigma_c) &= \prod_{c^j \in C} p(c^j|\mu_c, \Sigma_c), \\ p(Z|C) &= \prod_{t_j \in T} \prod_{p=1}^{L} discrete(logistic(c^j)), \\ p(V|Z) &= \prod_{t_j \in T} \prod_{p=1}^{L} \beta_{z_p^j}, \\ p(S|WC^T, \tau) &= \prod_{t_j \in T} \prod_{a_{ij}=1} p(s_{ij}|w^i (c^j)^T, \tau), \end{aligned}$$

and $p(w^i|\mu_w, \Sigma_w)$, $p(c^j|\mu_c, \Sigma_c)$, $discrete(logistic(c_j))$, $\beta_{z_p^j}$, and $p(s_{ij}|w^i(c^j)^T, \tau)$ are defined from Equation 2 to Equation 6, respectively. For brevity, we omit the conditional part of the joint distribution $p(V, S|A, \varphi)$ and abbreviate it to $p(V, S)$ in the rest of this paper.

Based on the model, the problem of worker skill estimation problem can be transformed into a probabilistic inference problem, namely, finding the maximum a posterior (MAP) configuration of the worker skill W and latent task category C conditioning on the resolved tasks (V, S). That is to find

$$(W^*, C^*, Z^*) = \arg \max_{W,C,Z} p(W, C, Z|V, S) \tag{7}$$

where $p(W, C, Z|V, S)$ is the posterior distribution of W and C given collected resolved tasks (V, S). However, it is difficult to compute the joint posterior distribution of W and C,

$$p(W, C, Z|V, S) = \int_\varphi p(W, C, Z, \varphi|V, S) d\varphi \tag{8}$$

where

$$p(W, C, Z, \varphi|V, S) = \frac{p(W, C, Z, \varphi, V, S)}{\int_{W,C,Z,\varphi} p(W, C, Z, \varphi, V, S) dW dC dZ d\varphi}.$$

This distribution is intractable to compute due to the coupling between the model parameters in φ. To tackle this problem, we develop an efficient and effective approximation in the next section.

5. A VARIATIONAL ALGORITHM

In this section, we propose a variational algorithm to approximate the distribution $p(W, C, Z|V, S)$ defined in Equation 8. The basic idea of our variational algorithm is to approximate the distribution $p(W, C, Z|V, S)$ using a variational distribution $q(W, C, Z)$ that is tractable for the maximization over W, C and Z in Equation 7.

5.1 Family of Variational Distributions

We restrict the variational distribution to a family of distributions that factorize as follows:

$$q(W, C, Z) = \prod_{i=1}^{M} q(w^i) \prod_{j=1}^{N} (q(c^j) \prod_{p=1}^{L} q(z_p^j)).$$

Then, we further require the distribution in this family to take the following parametric form:

$$\begin{aligned} & q(W, C, Z|\lambda_w, \nu_w, \lambda_c, \nu_c, \phi) \\ = & \prod_{i=1}^{M} q(w^i|\lambda_w^i, diag((\nu_w^i)^2)) \prod_{j=1}^{N} (q(c^j|\lambda_c^j, diag((\nu_c^j)^2)) \\ \times & \prod_{p=1}^{L} q(z_p^j|\phi_p^j)), \end{aligned}$$

where

$$\begin{aligned} q(w^i|\lambda_w^i, diag((\nu_w^i)^2)) &= Normal(\lambda_w^i, diag((\nu_w^i)^2)), \\ q(c^j|\lambda_c^j, diag((\nu_c^j)^2)) &= Normal(\lambda_c^j, diag((\nu_c^j)^2)), \\ q(z_p^j|\phi_p^j) &= discrete(\phi_p^j). \end{aligned}$$

$diag(\cdot)$ is a diagonal matrix where the entries outside the main diagonal are all zero. Here λ_w^i, $diag((\nu_w^i)^2)$, λ_c^j, $diag((\nu_c^j)^2)$, ϕ_p^j are variational parameters. For brevity, we denote the collection of variational parameters as $\varphi' = \{\lambda_w, diag(\nu_w^2), \lambda_c, diag(\nu_c^2), \phi\}$

in the rest of the paper. Thus, the inference for W, C, and Z in Equation 7 can be simplified as follows:

$$\begin{aligned}
&(W^*, C^*, Z^*)\\
=\ &[\arg\max_{w^1} q(w^1), \ldots, \arg\max_{w^M} q(w^M),\\
&\arg\max_{c^1} q(c^1), \ldots, \arg\max_{c^N} q(c^N),\\
&\arg\max_{z^1} q(z^1), \ldots, \arg\max_{z^N} q(z^N)]\\
=\ &[\arg\max_{\lambda_w^1, diag((\nu_w^1)^2)} q(w^1), \ldots, \arg\max_{\lambda_w^M, diag((\nu_w^M)^2)} q(w^M),\\
&\arg\max_{\lambda_c^1, diag((\nu_c^1)^2)} q(c^1), \ldots, \arg\max_{\lambda_c^N, diag((\nu_c^N)^2)} q(c^N),\\
&\arg\max_{\phi^1} q(z^1), \ldots, \arg\max_{\phi^N} q(z^N)].
\end{aligned}$$

5.2 Stationary Points of $L(q)$

The goal of the variational algorithm is to find the variational distribution that is close to the true posterior $p(W, C, Z|V, S)$. This is equivalent to optimizing the variational parameters φ' with respect to some distance measure, given by

$$\varphi' = \arg\max_{\varphi'} D(q(\varphi')||p(W, C, Z|V, S))). \qquad (9)$$

In this work, we adopt the Kullback-Leibler (KL) divergence which is commonly used to measure the difference between two distributions. It is defined as

$$KL(q||p) = \int_{\varphi'} q(\varphi') \log \frac{q(\varphi')}{p(W, C, Z|V, S)} d\varphi'$$

where KL divergence is a function of the variational parameters λ_w, $diag(\nu_w^2)$, λ_c, $diag(\nu_c^2)$, ϕ. However, directly optimizing the KL divergence is infeasible because the KL divergence involves the term $p(W, C, Z|V, S)$, which is intractable.

Instead, we solve an equivalent maximization problem, whose objective function is defined as

$$\begin{aligned}
\mathcal{L}(q) &= \int_{\varphi'} q(\varphi') \log \frac{p(W, C, Z, V, S)}{q(\varphi')} d\varphi'\\
&= E_q[\log p(W|\mu_w, diag(\nu_w^2))] + E_q[\log p(V|Z, \beta)]\\
&+ E_q[\log p(C|\mu_c, diag(\nu_c^2))] + E_q[\log p(Z|C)]\\
&- E_q[\log p(W|\lambda_w, diag(\nu_w^2))] - E_q[\log p(Z|\phi)]\\
&- E_q[\log p(C|\lambda_c, diag(\nu_c^2)] + E_q[\log p(S|WC^T, \tau)].
\end{aligned}$$

The expectations are taken with respect to the variational distribution $q(\cdot|\cdot)$ and subscripts denote the variational parameters involved in the expressions.

The equivalent between these two optimization problem can easily be seen as their objective functions sum up to a constant

$$KL(q||p) + \mathcal{L}(q) = \log p(V, S).$$

However, it is difficult to compute the expected log probability of $p(Z|C)$ in Equation 5. To preserve the lower bound of $\mathcal{L}$(q), we upper bound the log normalize with a Taylor expansion, given by

$$\begin{aligned}
E_q[\log p(Z|C)] &= E_q[Z^T C] - E_q[\log(\sum_{k=1}^{K} \exp\{C_k\})]\\
&\geq E_q[Z^T C] - \varepsilon^{-1}(\sum_{k=1}^{K} E_q[\exp\{C_k\}])\\
&+ 1 - \log \varepsilon = E_q'[\log p(Z|C)].
\end{aligned}$$

where ε is a new variational parameter. Thus, we replace the term $E_q[\log p(Z|C)]$ with $E_q'[\log p(Z|C)]$ in $\mathcal{L}(q)$ and obtain a lower bound of $\mathcal{L}(q)$, denoted as $\mathcal{L}'(q)$.

In order to maximize the objective function $\mathcal{L}(q)$, we take the derivatives of its lower bound $\mathcal{L}'(q)$ with respect to the variational parameters λ_w, ν_w, λ_c, ν_c, ϕ, ε, and set these derivatives to zeros.

$$\nabla_{\varphi'} \mathcal{L}'(q) = (\frac{\partial \mathcal{L}'(q)}{\partial \lambda_w}, \frac{\partial \mathcal{L}'(q)}{\partial \nu_w}, \frac{\partial \mathcal{L}'(q)}{\partial \lambda_c}, \frac{\partial \mathcal{L}'(q)}{\partial \nu_c}, \frac{\partial \mathcal{L}'(q)}{\partial \phi}, \frac{\partial \mathcal{L}'(q)}{\partial \varepsilon}) = \overrightarrow{0}.$$

For clarity, we put all the derivations in Appendix 10.1. We report the solutions to the optimization problem by

$$\begin{aligned}
\lambda_w^i &= (\Sigma_w^{-1} + \frac{1}{\tau^2} \sum_{t_j: a_{ij}=1} ((\lambda_c^j)^T \lambda_c^j + diag((\nu_c^j)^2)))^{-1}\\
&\times (\Sigma_w^{-1} \mu_w + \frac{1}{\tau^2} \sum_{t_j: a_{ij}=1} s_{ij} \lambda_c^j), \qquad (10)\\
(\nu_{w,k}^i)^2 &= (\sum_{t_j: a_{ij}=1} \frac{(\lambda_{c,k}^j)^2 + (\nu_{c,k}^j)^2}{\tau^2} + \Sigma_{w,kk}^{-1})^{-1}, \qquad (11)\\
\phi_p^j &\propto \exp(\lambda_{c,v}^j + \frac{1}{L} \sum_{j=1}^{T} \sum_{p=1}^{L} 1[v_p^j = v] \log \beta_{c,v}), \qquad (12)\\
\varepsilon_j &= \sum_{k=1}^{K} \exp(\lambda_{c,k}^j + \frac{(\nu_{c,k}^j)^2}{2}), \qquad (13)
\end{aligned}$$

for all $i = 1, \ldots, M$, $k = 1, \ldots, K$, and $v = 1, \ldots, V$.

However, $\mathcal{L}'(q)$ is not amenable to analytic maximization with respect to λ_c^j and ν_c. Thus, we use the conjugate gradient algorithm with derivative

$$\begin{aligned}
\frac{\partial \mathcal{L}'(q)}{\partial \lambda_c^j} &= (\Sigma_c^{-1} + \frac{1}{\tau^2} \sum_{w^i: a_{ij}=1} (\lambda_w^i (\lambda_w^i)^T + diag((\nu_w^i)^2))) \lambda_c^j\\
&+ \Sigma_c^{-1} \mu_c + \frac{1}{\tau^2} \sum_{w^i: a_{ij}=1} s_{ij} \lambda_w^i + L\phi^j\\
&- \frac{1}{\varepsilon_j} \exp\{\lambda_c^j + \frac{(\nu_c^j)^2}{2}\} \qquad (14)
\end{aligned}$$

$$\begin{aligned}
\frac{\partial \mathcal{L}'(q)}{\partial \nu_{c,k}^j} &= (\sum_{w^i: a_{ij}=1} \frac{(\lambda_{w,k}^i)^2 + (\nu_{w,k}^i)^2}{\tau^2} + \Sigma_{c,kk}^{-1})(\nu_c^j)^2\\
&- \frac{1}{\varepsilon_j} \exp\{\lambda_c^j + \frac{(\nu_c^j)^2}{2}\} \qquad (15)
\end{aligned}$$

for all $j = 1, \ldots, N$ and $k = 1, \ldots, K$.

5.3 Optimization Procedure

Based on the estimated stationary points, we optimize the model parameters $\varphi = \{\mu_w, \Sigma_w, \mu_c, \Sigma_c, \tau, \beta_{c,v}\}$ in this section.

We take the derivative of the lower bound $\mathcal{L}'(q)$ with respect to the model parameters φ, and set these derivatives to zeros.

$$\nabla_{\varphi} \mathcal{L}'(q) = (\frac{\partial \mathcal{L}'(q)}{\partial \mu_w}, \frac{\partial \mathcal{L}'(q)}{\partial \Sigma_w}, \frac{\partial \mathcal{L}'(q)}{\partial \mu_c}, \frac{\partial \mathcal{L}'(q)}{\partial \Sigma_c}, \frac{\partial \mathcal{L}'(q)}{\partial \tau}, \frac{\partial \mathcal{L}'(q)}{\partial \beta_{c,v}}) = \overrightarrow{0}.$$

For clarity, we put all the derivations in Appendix 10.2. We report the solutions to the optimization problem by

Algorithm 2 Iterative Optimization Algorithm

Input: a set of task assignments A, a set of resolved tasks (W, T, R),a limit on the number of iterations n_{max}
Output: variational parameters φ' and model parameters φ

```
1: $n \leftarrow 0$
2: **repeat**
3:     (a) Given $\varphi$, update $\varphi'$ according to Equations (10)-(15)
4:     (b) Given $\varphi'$, update $\varphi$ according to Equations (16)-(21)
5:     $n \leftarrow n + 1$
6: **until** $\mathcal{L}'(q^{(n)}) - \mathcal{L}'(q^{(n-1)}) \leq \epsilon$ or $n > n_{max}$
7: **return** $\varphi'$ and $\varphi$
```

$$\mu_w = \frac{1}{M}\sum_{i=1}^{M}\lambda_w^i \tag{16}$$

$$\Sigma_w = \frac{1}{M}\sum_{i=1}^{M}(diag((\nu_w^i)^2) + (\lambda_w - \mu_w)(\lambda_w - \mu_w)^T) \tag{17}$$

$$\mu_c = \frac{1}{N}\sum_{j=1}^{N}\lambda_c^j \tag{18}$$

$$\Sigma_c = \frac{1}{N}\sum_{j=1}^{N}(diag((\nu_c^j)^2) + (\lambda_c - \mu_c)(\lambda_c - \mu_c)^T) \tag{19}$$

$$\begin{aligned}\tau^2 &= \frac{1}{|A|}\sum_{a_{ij}=1}\{s_{ij}^2 + (\lambda_w^i)^T diag((\nu_c^j)^2)\lambda_w^i \\ &+ (\lambda_c^j)^T diag((\nu_w^i)^2)\lambda_c^j - 2s_{ij}(\lambda_w^i)^T\lambda_c^j \\ &+ ((\lambda_w^i)^T\lambda_c^j)^2 + Tr(diag((\nu_w^i)^2)diag((\nu_c^j)^2))\}\end{aligned} \tag{20}$$

$$\beta_{k,v} \propto \sum_{j=1}^{T}\sum_{p=1}^{L}\phi_{p,k}^j 1(v_p^j = v) \tag{21}$$

for all $s = 1, \ldots, K$ and $v = 1, \ldots, V$.

We present our optimization method in Algorithm 2. Algorithm 2 iteratively updates the variational parameters φ' and model parameters φ in Lines 3 and 4 until the objective function becomes convergent. The objective function $\mathcal{L}'(q)$ can be persistently improved by variational algorithm, stated in [22]. Because the value of $\mathcal{L}'(q)$ is finite, Algorithm 2 is guaranteed to converge with a finite number of iterations.

6. CROWD-SELECTION ALGORITHM

In this section, we introduce our crowd-selection algorithm based on bayesian model.

Given a new crowdsourced task t_j, we want to estimate its latent category such that the workers skilled in c^j can be selected. We first compute its variational parameters λ_c^j and ν_c^j under our bayesian model with parameters $\{\mu_c, \Sigma_c, \beta_{k,v}\}$. The estimation procedure is similar to Algorithm 2, but the terms depending on worker skill and feedback scores are removed. Thus, we compute its latent category using conjugate gradient algorithm with derivative

$$\frac{\partial\mathcal{L}'(q)}{\partial\lambda_c^j} = \Sigma_c^{-1}\lambda_c^j + \Sigma_c^{-1}\mu_c + L\phi^j - \frac{1}{\varepsilon_j}\exp\{\lambda_c^j + \frac{(\nu_c^j)^2}{2}\} \tag{22}$$

$$\frac{\partial\mathcal{L}'(q)}{\partial\nu_{c,k}^j} = \Sigma_{c,kk}^{-1}(\nu_c^j)^2 - \frac{1}{\varepsilon_j}\exp\{\lambda_c^j + \frac{(\nu_c^j)^2}{2}\} \tag{23}$$

Algorithm 3 Task-Driven Crowd-Selection Algorithm

Input: A task t_j, worker skill W, model parameters φ, number of workers k, a limit on the number of iterations n_{max}
Output: A set of selected workers R

```
1: $n \leftarrow 0$
2: **repeat**
3:     (a) Update variational $\lambda_c^j$ and $\nu_c^j$ by Equations (22)-(23)
4:     (b) Update $\phi^j$ and $\varepsilon_j$ by Equations (12)-(13)
5: **until** $n > n_{max}$
6: Sample $c^j \sim Normal(\lambda_c^j, \nu_c^j)$
7: Choose selected workers $R$ by Equation 1
8: **return** selected workers $R$
```

and the update for ϕ^j and ε_j are given in Equations 12 and 13.

We now present the details of the task-driven crowd-selection in Algorithm 3. Given a new task t_j and model parameters φ, Algorithm 3 selects top-k workers R for this task. In the first phrase, Algorithm 3 computes the variational parameters λ_c^j and ν_c^j for the latent category of task t_j from Line 2 to Line 5. The latent category for task t_j is sampled by a Normal distribution with mean λ_c^j and variance ν_c^j in Line 6. In the second phrase, Algorithm 3 chooses a set of workers R based on worker skill W and latent category c^j by Equation 1 in Line 7. Finally, Algorithm 3 returns the selected workers R in Line 8.

7. EXPERIMENTAL STUDY

In this section, we evaluate the performance of our algorithms. All the algorithms, including those we compared within the experiments, were implemented in Java (we will release all our source codes if the paper is published) and tested on machines with Windows OS, Intel(R) Core(TM2) Quad CPU 2.66Hz, and 60GB of RAM memory.

7.1 Datasets

We collect the data from three well-known crowdsourcing applications Quora, Yahoo ! Answer and Stack Overflow. Some statistics of the datasets are reported in Table 2.

7.1.1 Quora

We gathered our Quora dataset through web-based crawls between August and early September 2012. We limited these crawls to 10 requests/second to minimize the impact on Quora[4].

We start our crawls using 100 randomly selected questions. The crawls follow a BFS pattern through the related questions link for each question. In total, we collect 444,000+ unique questions. Each question page contains a complete list of answers, and the respondent and voters for each answer. As shown in Table 2, this question-based crawl produced 444,000+ unique questions, 887,000+ unique answers, and 95,000+ unique users who answered a question. For each answer, we consider the number of thumbs-up voted by the crowd as the quality measure.

7.1.2 Yahoo ! Answer

We collect our Yahoo ! Answer dataset[5] through its API from Jan, 2012 onwards. In total, we collect 8866,000+ unique questions. On the average, each question has around three respondents.

[4]`http://www.quora.com/`
[5]`http://answers.yahoo.com/`

Table 2: Statistics of Real Datasets

Dataset	Total Questions	Total Users	Total Answers
Quora	444k	95k	887k
Yahoo ! Answer	8866k	1004k	26903k
Stack Overflow	83k	15k	236k

We also crawl the best answer for each question from Yahoo ! Answer, which is used to mark the best answerer. As shown in Table 2, the API-based crawl produced 8866,000+ unique questions, 26903,000+ unique answers, and 1004,000+ unique users who answered a question.

7.1.3 *Stack Overflow*

We download the Stack Overflow dataset from the website[6]. This dataset containing the questions and the related answerers posted between February 18, 2009 and June 7, 2009. For each answer of a question, the Stack Overflow provides the score of the answer. We consider the answer with the highest score as the best answer. The statistics of the dataset is given in Table 2.

7.2 Experimental Settings

We detail the experimental settings in this subsection, including the algorithm for comparison, the measures we use to assess the performance of the algorithms.

7.2.1 *Algorithms for Comparison*

We compare our task-driven crowd-selection algorithm, denoted as **TDPM** (Task-Driven Probabilistic Model), to several state-of-the-art algorithms such as Vector Space Model (VSM), Dual Role Model (DRM) [28] and Topic Sensitive Probabilistic Model (TSPM) [8].

- **VSM.** The VSM algorithm selects the workers based on the cosine similarity between the crowdsourced task and the historical tasks resolved by the workers. Consider t^j as a bag of vocabularies for the given task defined in Section 4.1. Then, we define t^i_w as a bag of vocabularies of the task resolved by worker w^i where $t^i_w = \bigcup_{t^j:a_{ij}=1} t^j$. The ranking score of worker w^i is given by
$$\hat{s_{ij}} = \frac{(t^j)^T t^i_w}{\sqrt{(t^j)^T t^j}\sqrt{(t^i_w)^T t^i_w}}.$$
where the numerator $(t^j)^T t^i_w$ is a dot product of vocabulary vectors t^j and t^i_w.

- **DRM.** The DRM algorithm models the skills of workers as a Multinomial distribution. Then it carries out the estimation using **Probabilistic Latent Semantic Analysis** [10] for skills of worker w^i as well as the latent category of the task c^j. Given a crowdsourced task t^j, the crowd-selection of DRM on workers is proportional to the predictive performance $w^i(c^j)^T$.

- **TSPM.** The TSPM algorithm also models the skills of workers as a Multinomial distribution. The estimation of worker skill w^i and latent category of the task c^j are based on **Latent Dirichlet Allocation**. Similarly, the crowd-selection of TSPM on workers is based on the predictive score of workers on task t^j which is $w^i(c^j)^T$.

[6]`http://www.ics.uci.edu/~duboisc/stackoverflow/`

However, the sum of weights on the skills of all workers are normalized to one (i.e. $\sum_{k=1}^K w^i_k = 1$) because the property of Multinomial distribution. We argue that their models can not compare the skills of workers on a specific latent category (i.e. $w^i_k > w^j_k$?). We devise another model-based approach to tackle this problem.

7.2.2 *Job Quality Assessment*

We assess the performance of our algorithm by two measurements: *precision* and *recall*.

We employ the formula *ACCU* to measure the precision of the crowd-selection algorithms, which is also used in DRM [28] and TSPM [33]. ACCU is defined as the ratio of the rank of the right worker to the total number of candidate workers. The formula of ACCU is given by

$$ACCU = \frac{|R| - R_{best} - 1}{|R| - 1}$$

where R is the set of selected workers and R_{best} is the rank of right worker in R. We consider the rank of the best answerer as R_{best} in Yahoo ! Answer while we regard the rank of the worker with highest score as R_{best} in Quora and Stack Overflow.

We propose to use *TopK* to measure the recall of the crowd-selection algorithm. The TopK is defined as the ratio of the number of times that the rank of right worker is less than K to the total number of crowdsourced tasks N. The formula of TopK is given by

$$TopK = \frac{|\{t^j|R^j_{best} \leq K\}|}{N}.$$

In this experiment, we evaluate the recall of the crowd-selection algorithms using Top1 and Top2.

7.3 Performance Results

We report and discuss the performance results of our TDPM, with VSM, TSPM and DRM, for each of the three datasets as follows.

7.3.1 *Performance on Quora*

For Quora dataset, we first extract the group of workers based on their participation in solving tasks. We denote the group of workers who solve more than n tasks in Quora as $Quora_n$. For example, $Quora_3$ is a group of workers who solve more than three tasks in Quora. $Quora_1$ consists of all the workers in Quora. In this experiment, we extract nine groups of workers for testing, denoted as $Quora_1$, $Quora_2$, ..., $Quora_9$.

To analyze the extracted groups, we define task coverage of a group to be the ratio of the number of distinct task solved to the total number of tasks. We illustrate task coverage of the groups by varying the task participation threshold in Figure 3(a). We also show the size of the groups by varying the task participation threshold in Figure 3(b). We can see that the task coverage of $Quora_5$ is above 0.92 while the number of workers in $Quora_5$ is around 30,000+ (only one third of the total workers). We can conclude the size of the group with high task participation threshold is small and the crowd-selection from these groups can achieve high task coverage.

Then, we test the performance of the crowd-selection algorithms on different groups. For fairness, we randomly choose 10k questions for each group where the right worker for each testing question must be in the group.

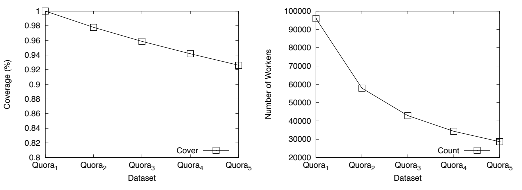
The running time of the algorithms in Quora for Top1 and Top2 crowd-selection are illustrated in Figures 4(a) and 4(b). We can see that the running time of all the algorithms increase with respect

Table 3: Precision of Crowd-Selection Algorithms in Quora (The Best Score in Bold)

Algorithm	$Quora_1$					$Quora_5$					$Quora_9$				
/Category	10	20	30	40	50	10	20	30	40	50	10	20	30	40	50
VSM	0.859					0.873					0.881				
TSPM	0.935	0.936	0.936	0.935	0.935	0.945	0.946	0.947	0.946	0.946	0.953	0.953	0.952	0.953	0.953
DRM	0.936	0.935	0.936	0.936	0.935	0.945	0.946	0.945	0.946	0.948	0.952	0.952	0.954	0.952	0.953
TDPM	**0.945**	**0.948**	**0.950**	**0.951**	**0.951**	**0.957**	**0.959**	**0.961**	**0.961**	**0.962**	**0.962**	**0.965**	**0.966**	**0.966**	**0.966**

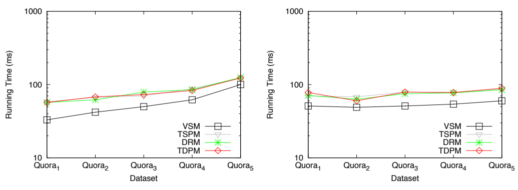
Table 4: Recall of Crowd-Selection Algorithms in Quora (The Best Score in Bold)

Algorithm	$Quora_1$		$Quora_2$		$Quora_3$		$Quora_4$		$Quora_5$	
/TopK	1	2	1	2	1	2	1	2	1	2
VSM	0.733	0.887	0.737	0.891	0.74	0.894	0.743	0.897	0.745	0.899
TSPM	0.882	0.957	0.866	0.939	0.848	0.918	0.831	0.9	0.814	0.882
DRM	0.882	0.956	0.866	0.937	0.844	0.916	0.829	0.9	0.815	0.883
TDPM	**0.8906**	**0.963**	**0.877**	**0.944**	**0.868**	**0.928**	**0.852**	**0.912**	**0.852**	**0.912**



(a) Task Coverage (b) Group Size

Figure 3: Statistics of the Crowd in Quora



(a) Top1 Crowd-Selection (b) Top2 Crowd-Selection

Figure 4: Running Time of Crowd-Selection Algorithms in Quora

to the groups of workers who participate more. This is because the questions answered by active workers are usually popular and attract more workers. Thus, the running cost for selecting right workers increases. We also observe that the running time increases slowly for DRM, TSPM and our TDPM since the estimation of latent category of the given task is the main computation cost.

Now, we investigate the effectiveness of the crowd-selection algorithms. For the Quora dataset, we set the number of latent task category k = 10, 20, 30, 40, and 50, respectively. We demonstrate the precision of the crowd-selection algorithms in Table 5 and the recall in Table 6.

We show the precision of the crowd-selection algorithm on three groups $Quora_1$, $Quora_5$ and $Quora_9$ by varying the number of latent categories from 10 to 50 in Table 5. We can observe that the precision of our algorithm is more superior to the other three algorithms on all the number of latent categories set. We can also find that the precision of all the algorithms increases when we select the crowd from more active workers (i.e. we select the workers from the group $Quora_n$). Then, we conclude that the active workers are usually the providers of the best answers. For all the algorithms based on the latent category, we notice that the precision increases and then becomes convergent when we add the number of latent categories.

We illustrate the recall of the algorithms for Top1 and Top2 Crowd-Selection on five groups in Table 6. We can see that both Top1 and Top2 recall of our algorithm is superior than other algorithms for all groups tested. We can also notice that the TopK recall of all the algorithms decreases with respect to the groups of workers who participate more. As mentioned, the questions answered by active workers are popular and the number of workers increases. Thus, the Topk crowd-selection becomes uncertain and the TopK recall of all algorithms decreases in Table 6.

7.3.2 Performance on Yahoo ! Answer

For Yahoo ! Answer dataset, we extract five groups of workers for testing, denoted as $Yahoo_{10}$, $Yahoo_{15}$, . . ., $Yahoo_{30}$.

We illustrate task coverage of the groups by varying the task participation threshold in Figure 5(a). We also demonstrate the size of the groups by varying the task participation threshold in Figure 5(b). We can see that the task coverage of $Yahoo_{30}$ is around 0.93 while the number of workers in $Yahoo_{30}$ is around 100k (only one tenth of the total workers).

We also randomly choose 10k questions for testing. The running time of the algorithms in Yahoo ! Answer for Top1 and Top2 crowd-selection are illustrated in Figures 6(a) and 6(b). We can see that the running time of all algorithms increase for the questions answered by more active workers.

Now, we validate the precision of the crowd-selection algorithms in Table 5 and the recall in Table 6. We report the precision of the crowd-selection algorithms on $Yahoo_{10}$, $Yahoo_{15}$ and $Yahoo_{20}$ with respect to the number of latent categories. Compared with Quora, we find the precision of the crowd-selection algorithms on Yahoo ! Answer converges faster on the number of latent categories. The precision of VSM is much lower in Table 5. This is because the questions in Yahoo ! Answer are very short compared with the questions in Quora. We show the Top1 and Top2 recall of the crowd-selection algorithms on $Yahoo_{10}$, $Yahoo_{15}$, . . ., $Yahoo_{30}$ in Table 6. Both Top1 and Top2 recall of the crowd-selection algorithm decrease for the questions selected for more active workers.

7.3.3 Performance on Stack Overflow

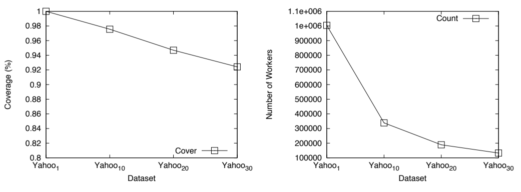
For Stack Overflow dataset, we extract five groups of workers for testing denoted as $Stack_1$, $Stack_3$, . . ., $Stack_{12}$.

Table 5: Precision of Crowd-Selection Algorithms in Yahoo ! Answer (The Best Score in Bold)

Algorithm	$Yahoo_{10}$					$Yahoo_{15}$					$Yahoo_{20}$				
/Category	10	20	30	40	50	10	20	30	40	50	10	20	30	40	50
VSM	0.665					0.676					0.685				
TSPM	0.855	0.849	0.848	0.847	0.847	0.884	0.879	0.878	0.877	0.877	0.905	0.900	0.899	0.899	0.899
DRM	0.846	0.847	0.847	0.847	0.847	0.877	0.878	0.877	0.877	0.877	0.898	0.900	0.8992	0.900	0.898
TDPM	**0.945**	**0.947**	**0.948**	**0.949**	**0.949**	**0.965**	**0.969**	**0.970**	**0.969**	**0.970**	**0.981**	**0.984**	**0.984**	**0.984**	**0.985**

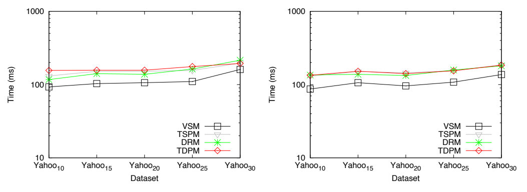
Table 6: Recall of Crowd-Selection Algorithms in Yahoo ! Answer (The Best Score in Bold)

Algorithm	$Yahoo_{10}$		$Yahoo_{15}$		$Yahoo_{20}$		$Yahoo_{25}$		$Yahoo_{30}$	
/TopK	1	2	1	2	1	2	1	2	1	2
VSM	0.518	0.721	0.515	0.717	0.511	0.708	0.504	0.702	0.496	0.693
TSPM	0.695	0.833	0.655	0.810	0.644	0.805	0.637	0.802	0.626	0.795
DRM	0.655	0.823	0.636	0.815	0.629	0.811	0.628	0.809	0.626	0.809
TDPM	**0.823**	**0.908**	**0.815**	**0.904**	**0.811**	**0.903**	**0.809**	**0.901**	**0.809**	**0.901**



(a) Task Coverage (b) Group Size

Figure 5: Statistics of the Crowd in Yahoo ! Answer



(a) Top1 Crowd-Selection (b) Top2 Crowd-Selection

Figure 6: Running Time of Crowd-Selection Algorithms in Yahoo ! Answer

We show task coverage of the groups by varying the task participation threshold in Figure 7(a). We illustrate the number of workers in the groups by varying the task participation threshold in Figure 7(b). We can find that the task cover of $Stack_{12}$ is around 0.9 while the number of workers in $Stack_{12}$ is less than 5k (only one sixth of the total workers).

We randomly choose 1k questions for testing. The running time of the algorithms in Stack Overflow for Top1 and Top2 crowd-selection are illustrated in Figures 8(a) and 8(b). We can see that the running time of all algorithm increase more rapidly than the running time in Quora and Yahoo ! Answer. This is because that there are more workers providing answers for popular questions in Stack Overflow.

We now show the precision of the crowd-selection algorithms in Table 7. We observe that the precision of crowd-selection algorithms increases quickly when we select active workers. We conclude that this is because the users in Stack Overflow usually trust the workers with high reputation such that the active workers are more likely to get best score. We also find that the precision of VSM algorithm is competitive in Table 7. This is because that we use the wisely labeled tags of the questions for vocabularies of the tasks. We illustrate the Top1 and Top2 recall of the crowd-selection algorithms in Table 8. We can notice that the Top1 and Top2 recall of crowd-selection algorithms in Stack Overflow is lower than the recall in other two datasets when we select the workers for more popular questions. We argue that this is because the popular questions in Stack Overflow attract more workers to solve them.

7.3.4 *Conclusion on Performance Comparison*

In conclusion, the results in Sections 7.3.1, 7.3.2, and 7.3.3 show that TDPM consistently attains high crowd-selection quality in terms of both precision and recall. Compared with the state-of-the-art algorithms, our TDPM algorithm benefits from two aspects new worker skill model and task feedback score. The new worker skill model makes the skills of workers on latent task categories become comparable. We also utilize the task feedback score which can be widely collected in many crowdsourcing platforms to further improve the estimation of the skills of workers on latent categories. In this experimental study, we also find that the crowd-selection from active workers also can greatly improve the precision.

8. CONCLUSION

We studied the problem of task-driven crowd-selection for crowdsourced tasks. Unlike the existing works based on the trustworthiness, our work is to devise a bayesian model that exploits "who knows what" for the workers in crowdsourcing system. The model builds the latent category space of the crowdsourced tasks as well as infers the latent skills of workers on the space. The probabilistic inference for the proposed bayesian model is based on the feedback scores of the past resolved tasks. We then develop a variational algorithm that transforms the probabilistic inference into a standard optimization problem, which can be solved efficiently. We also devise an incremental crowd-selection algorithm that projects the coming tasks into the existing latent category space and choose the highly skilled workers for the tasks. We validate the performance of our algorithm based on the data collected from three well-known crowdsourcing applications: Quora, Yahoo ! Answer and Stack Overflow.

ACKNOWLEDGEMENTS This work is partially supported by GRF under grant number HKUST 617610.

9. REFERENCES

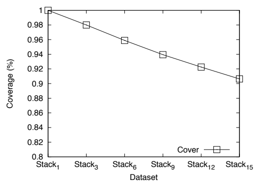
[1] A. Bosu, C. S. Corley, D. Heaton, D. Chatterji, J. C. Carver, and N. A. Kraft. Building reputation in stackoverflow: an

Table 7: Precision of Crowd-Selection Algorithms in Stack Overflow (The Best Score in Bold)

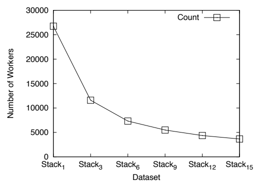
Algorithm	$Stack_1$					$Stack_6$					$Stack_{12}$				
/Category	10	20	30	40	50	10	20	30	40	50	10	20	30	40	50
VSM	0.693					0.738					0.758				
TSPM	0.711	0.706	0.704	0.702	0.702	0.851	0.847	0.849	0.849	0.847	0.935	0.932	0.932	0.931	0.931
DRM	0.710	0.708	0.706	0.708	0.707	0.854	0.850	0.852	0.850	0.849	0.936	0.934	0.933	0.931	0.932
TDPM	**0.801**	**0.813**	**0.817**	**0.821**	**0.824**	**0.930**	**0.941**	**0.946**	**0.950**	**0.954**	**0.997**	**1**	**1**	**1**	**1**

Table 8: Recall of Crowd-Selection Algorithms in Yahoo ! Answer (The Best Score in Bold)

Algorithm	$Stack_1$		$Stack_3$		$Stack_6$		$Stack_9$		$Stack_{12}$	
/TopK	1	2	1	2	1	2	1	2	1	2
VSM	0.433	0.685	0.465	0.713	0.483	0.727	0.493	0.739	0.502	0.749
TSPM	0.461	0.69	0.481	0.713	0.492	0.711	0.488	0.693	0.485	0.684
DRM	0.455	0.685	0.479	0.704	0.483	0.700	0.482	0.694	0.480	0.677
TDPM	**0.581**	**0.819**	**0.634**	**0.835**	**0.651**	**0.826**	**0.651**	**0.806**	**0.64**	**0.782**

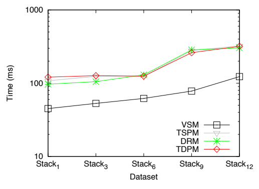


(a) Task Coverage

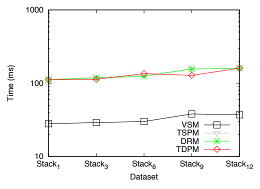


(b) Group Size

Figure 7: Statistics of the Crowd in Stack Overflow



(a) Top1 Crowd-Selection



(b) Top2 Crowd-Selection

Figure 8: Running Time of Crowd-Selection Algorithms in S-tack Overflow

empirical investigation. In *Proceedings of the Tenth International Workshop on Mining Software Repositories*, pages 89–92. IEEE Press, 2013.

[2] C. C. Cao, J. She, Y. Tong, and L. Chen. Whom to ask?: jury selection for decision making tasks on micro-blog services. *Proceedings of PVLDB*, 5(11):1495–1506, 2012.

[3] C. C. Cao, Y. Tong, L. Chen, and H. Jagadish. Wisemarket: a new paradigm for managing wisdom of online social users. In *Proceedings of SIGKDD*, pages 455–463. ACM, 2013.

[4] S. B. Davidson, S. Khanna, T. Milo, and S. Roy. Using the crowd for top-k and group-by queries. In *Proceedings of ICDT*, pages 225–236. ACM, 2013.

[5] G. Dror, Y. Koren, Y. Maarek, and I. Szpektor. I want to answer; who has a question?: Yahoo! answers recommender system. In *Proceedings of SIGKDD*, pages 1109–1117. ACM, 2011.

[6] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. Crowddb: answering queries with crowdsourcing. In *Proceedings of SIGMOD*, pages 61–72. ACM, 2011.

[7] J. Gao, X. Liu, B. C. Ooi, H. Wang, and G. Chen. An online cost sensitive decision-making method in crowdsourcing systems. In *SIGMOD*, 2013.

[8] J. Guo, S. Xu, S. Bao, and Y. Yu. Tapping on the potential of q&a community by recommending answer providers. In *CIKM*, pages 921–930. ACM, 2008.

[9] S. Guo, A. Parameswaran, and H. Garcia-Molina. So who won?: dynamic max discovery with the crowd. In *Proceedings of SIGMOD*, pages 385–396. ACM, 2012.

[10] T. Hofmann. Probabilistic latent semantic indexing. In *SIGIR*, pages 50–57. ACM, 1999.

[11] H. Kaplan, I. Lotosh, T. Milo, and S. Novgorodov. Answering planning queries with the crowd.

[12] X. Liu, M. Lu, B. C. Ooi, Y. Shen, S. Wu, and M. Zhang. Cdas: a crowdsourcing data analytics system. *Proceedings of PVLDB*, 5(10):1040–1051, 2012.

[13] A. Marcus, D. Karger, S. Madden, R. Miller, and S. Oh. Counting with the crowd. In *VLDB*, pages 109–120. VLDB Endowment, 2012.

[14] A. Marcus, E. Wu, D. Karger, S. Madden, and R. Miller. Human-powered sorts and joins. *Proceedings of PVLDB*, 5(1):13–24, 2011.

[15] A. Parameswaran, A. D. Sarma, H. Garcia-Molina, N. Polyzotis, and J. Widom. Human-assisted graph search: it's okay to ask questions. *Proceedings of PVLDB*, 4(5):267–278, 2011.

[16] A. G. Parameswaran, H. Garcia-Molina, H. Park, N. Polyzotis, A. Ramesh, and J. Widom. Crowdscreen: Algorithms for filtering data with humans. In *Proceedings of SIGMOD*, pages 361–372. ACM, 2012.

[17] H. Park and J. Widom. Query optimization over crowdsourced data, 2012.

[18] V. C. Raykar, S. Yu, L. H. Zhao, G. H. Valadez, C. Florin, L. Bogoni, and L. Moy. Learning from crowds. *JMRL*, 99:1297–1322, 2010.

[19] Y. Tian and J. Zhu. Learning from crowds in the presence of schools of thought. In *Proceedings of SIGKDD*, pages 226–234. ACM, 2012.

[20] B. Trushkowsky, T. Kraska, M. J. Franklin, and P. Sarkar. Crowdsourced enumeration queries. In *ICDE*, 2013.

[21] P. Venetis, H. Garcia-Molina, K. Huang, and N. Polyzotis. Max algorithms in crowdsourcing environments. In *Proceedings of WWW*, pages 989–998. ACM, 2012.

[22] M. J. Wainwright and M. I. Jordan. Graphical models,

exponential families, and variational inference. *Foundations and Trends® in Machine Learning*, 1(1-2):1–305, 2008.

[23] G. Wang, K. Gill, M. Mohanlal, H. Zheng, and B. Y. Zhao. Wisdom in the social crowd: an analysis of quora. In *Proceedings of WWW*, pages 1341–1352. International World Wide Web Conferences Steering Committee, 2013.

[24] J. Wang, T. Kraska, M. J. Franklin, and J. Feng. Crowder: Crowdsourcing entity resolution. *Proceedings of PVLDB*, 5(11):1483–1494, 2012.

[25] P. Welinder and P. Perona. Online crowdsourcing: rating annotators and obtaining cost-effective labels. In *CVPRW*, pages 25–32. IEEE, 2010.

[26] S. E. Whang, P. Lofgren, and H. Garcia-Molina. Question selection for crowd entity resolution. 2013.

[27] S. Wu, Z. Zhang, A. K. Tung, X. Wang, and S. Wang. K-anonymity for crowdsourcing database. *IEEE Transactions on Knowledge and Data Engineering*, page 1, 2013.

[28] F. Xu, Z. Ji, and B. Wang. Dual role model for question recommendation in community question answering. In *Proceedings of SIGIR*, pages 771–780. ACM, 2012.

[29] T. Yan, V. Kumar, and D. Ganesan. Crowdsearch: exploiting crowds for accurate real-time image search on mobile phones. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 77–90. ACM, 2010.

[30] C. J. Zhang, L. Chen, H. Jagadish, and C. C. Cao. Reducing uncertainty of schema matching via crowdsourcing. *VLDB*, 6(9), 2013.

[31] Z. Zhao, W. Ng, and Z. Zhang. Crowdseed: query processing on microblogs. In *Proceedings of EDBT*, pages 729–732. ACM, 2013.

[32] Z. Zhao, D. Yan, W. Ng, and S. Gao. A transfer learning based framework of crowd-selection on twitter. In *SIGKDD*, pages 1514–1517. ACM, 2013.

[33] G. Zhou, S. Lai, K. Liu, and J. Zhao. Topic-sensitive probabilistic model for expert finding in question answer communities. In *Proceedings of CIKM*, pages 1662–1666. ACM, 2012.

10. APPENDIX

10.1 Variational Parameter Estimation

We derive the partial derivatives one by one, starting with $\frac{\partial \mathcal{L}'}{\partial \lambda_w}$. For simplicity, we collect the terms involving λ_w in $\mathcal{L}'$ as

$$\begin{aligned}
\mathcal{L}'_{\lambda_w^i} &= E_q[\log p(W|\mu_w, diag(\nu_w^2))] + E_q[\log p(S|WC^T, \tau)] \\
&- E_q[\log p(W|\lambda_w, diag(\nu_w^2))] \\
&= \Sigma_w^{-1}\mu_w\lambda_w^i - \frac{1}{2}\Sigma_w^{-1}(\lambda_w^i)^T\lambda_w^i + \sum_{t^j:a_{ij}=1}\{\frac{1}{\tau^2}(\lambda_w^i)^T\lambda_c^j s_{ij} \\
&+ (\lambda_c^j)^T\lambda_w^i(\lambda_w^i)^T\lambda_c^j] - \frac{1}{2\tau^2}[(\lambda_w^i)^T\lambda_w^i diag((\nu_w^j)^2)\}.
\end{aligned}$$

Then, we set its derivative to zero and we obtain

$$\begin{aligned}
\lambda_w^i &= (\Sigma_w^{-1} + \frac{1}{\tau^2}\sum_{t^j:a_{ij}=1}((\lambda_c^j)^T\lambda_c^j + diag((\nu_c^j)^2)))^{-1} \\
&\times (\Sigma_w^{-1}\mu_w + \frac{1}{\tau^2}\sum_{t^j\in a_{ij}=1} s_{ij}\lambda_c^j).
\end{aligned}$$

For ν_w^i, we have

$$\begin{aligned}
\mathcal{L}_{\nu_{w,k}^i} &= E_q[\log p(W|\mu_w, diag(\nu_w^2))] + E_q[\log p(S|WC^T, \tau)] \\
&- E_q[\log p(W|\lambda_w, diag(\nu_w^2))] \\
&= -\frac{1}{2}\Sigma_w^{-1}diag((\nu_w^i)^2) - \sum_{t^j:a_{ij}=1}\{\frac{1}{2\tau^2}((\lambda_c^j)^T\lambda_c^j diag((\nu_w^i)^2) \\
&+ diag((\nu_w^i)^2)diag((\nu_c^j)^2)) - \frac{1}{2}\ln|diag((\nu_w^i)^2)|\}
\end{aligned}$$

By setting $\frac{\partial \mathcal{L}'}{\partial \nu_{w,k}^i} = 0$, we have

$$(\nu_{w,k}^i)^2 = (\sum_{t^j:a_{ij}}\frac{(\lambda_{c,k}^j)^2 + (\nu_{c,k}^j)^2}{\tau^2} + \Sigma_{w,kk}^{-1})^{-1}.$$

For $\phi_{c,v}$, we have

$$\begin{aligned}
\mathcal{L}'_{\phi_{c,v}} &= E_q[\log p(Z|C)] + E_q[\log p(V|Z,\beta)] - E_q[\log p(Z|\phi)]] \\
&= \lambda_c\phi + \phi\log\beta_{c,v} - \phi\log\phi.
\end{aligned}$$

By setting $\frac{\partial \mathcal{L}'}{\partial \phi_{c,v}} = 0$, we have

$$\phi_{c,v} \propto \exp(\lambda_{c,v}^j + \frac{1}{L}\sum_{j=1}^{T}\sum_{p=1}^{L}1[v_p^j = v]\log\beta_{c,v}).$$

For ε, we have

$$\mathcal{L}'_\varepsilon = -\varepsilon^{-1}(\sum_{k=1}^{K}E_q[\exp\{C_k\}]) - \log\varepsilon.$$

By setting $\frac{\partial \mathcal{L}'}{\partial \varepsilon} = 0$, we have

$$\varepsilon_j = \sum_{k=1}^{K}\exp(\lambda_{c,k}^j + \frac{(\nu_{c,k}^j)^2}{2})$$

10.2 Model Parameter Estimation

We estimate the model parameters based on the derived variational parameters.

We first maximize $\mathcal{L}'(q)$ with respect to μ_w, given by

$$\mathcal{L}'_{\mu_w} = \sum_{i=1}^{M}\Sigma_w^{-1}\mu_w\lambda_w^i - M\frac{1}{2}\mu_w^T\Sigma_w^{-1}\mu_w$$

Then, we set its derivative to zero and we obtain

$$\mu_w = \frac{1}{M}\sum_{i=1}^{M}\lambda_w^i.$$

For Σ_w, we have

$$\begin{aligned}
\mathcal{L}'_{\Sigma_w} &= \sum_{i=1}^{M}(\Sigma_w^{-1}\mu_w\lambda_w^i - \frac{1}{2}\Sigma_w^{-1}(diag((\nu_w^i)^2) + (\lambda_w^i)^T\lambda_w^i) \\
&- M(\frac{1}{2}\mu_w^T\Sigma_w^{-1}\mu_w + \frac{1}{2}\ln|\Sigma_w|)).
\end{aligned}$$

Next, we set its derivative to zero, given by

$$\begin{aligned}
\frac{\partial \mathcal{L}'}{\partial \Sigma_w} &= \sum_{i=1}^{M}(-\Sigma_w^{-T}\Sigma_w^{-T}\mu_w\lambda_w^i + \frac{1}{2}\Sigma_w^{-T}\Sigma_w^{-T}(diag((\nu_w^i)^2) + \lambda_w^i(\lambda_w^i)^T)) \\
&- M(-\frac{1}{2}\Sigma_w^T\mu_w\mu_w^T\Sigma_w^{-T} + \frac{1}{2}(\Sigma_w^T)^{-1}).
\end{aligned}$$

By setting $\frac{\partial \mathcal{L}'}{\partial \Sigma_w} = 0$, we have

$$\Sigma_w = \frac{1}{M}\sum_{i=1}^{M}(diag((\nu_w^i)^2) + (\lambda_w - \mu_w)(\lambda_w - \mu_w)^T).$$

The derivations for μ_c and Σ_c are similar to the above.

Dismantling Complicated Query Attributes with Crowd

Matan Laadan
Tel Aviv University
Tel Aviv, Israel
matanlaa@post.tau.ac.il

Tova Milo
Tel Aviv University
Tel Aviv, Israel
milo@cs.tau.ac.il

ABSTRACT

We study the problem of query evaluation with the help of the crowd, when the value of the queried attributes is not available in the database and is also hard for the crowd to estimate. Rather than asking users directly about these attributes, we propose a novel alternative approach that first uses the crowd to dismantle the query attributes into finer related ones (whose value estimation is easier), then assemble them to yield better estimation for the query attributes. We show that it is sometimes beneficial not to only dismantle the query attributes themselves, but rather to continue dismantling newly discovered attributes. We provide a careful statistical analysis to estimate the potential benefit (and cost) of dismantling each of the so-far-discovered attributes. Building on this analysis, we present an effective algorithm that balances between attributes dismantling and obtaining essential statistics about them (for estimating properties like "difficulty" and "contribution" of attributes) to decide how many crowd members should be asked about each attribute and how the answers should be assembled together. A thorough experimental analysis demonstrates the feasibility and effectiveness of the approach.

1. INTRODUCTION

We consider the problem of query evaluation with the help of the crowd, when the value of the query attributes is hard to estimate. Rather than asking users directly about these attributes, we propose a novel alternative approach that first uses the crowd to dismantle the query attributes into finer related ones (whose value estimation is easier), then assemble them to yield better estimation for the query attributes.

To illustrate, assume that we want to evaluate a query over a database of objects, testing and retrieving the values of certain attributes. This is a standard task when the attribute values are available explicitly in the database, but becomes challenging when they are not. For example, consider an imaginary cooking website CrowdCooking.com (CC) - a large recipes website where people can post their own recipes and other people can search and use them. Up until now, CC only allowed basic keyword search, but they now wish to upgrade their search capabilities to include more sophisticated searches, allowing people to search, e.g., for dessert recipes that are easy to make, have less than X calories and contain a certain amount of proteins. While NLP/text-analysis techniques could be used to evaluate some of these search criteria, this may be costly and inaccurate. An alternative approach that emerged in recent years is to use the crowd of web users for finding the value of the missing query attributes - Is a dish a dessert or not? How many calories/proteins does it contains? Etc. As crowd answers may be erroneous, a common approach is to ask multiple users about each missing query attribute and compute some aggregation (usually average/median) of the answers. The number of crowd members that need to be asked about a given attribute is typically determined by the difficulty of the question and the budget constraints[1]. For example, three users probably suffice to determine with a high probability that a certain dish is a dessert, but more are likely to be required to determine its number of calories. In fact, a key problem of this approach is that some attributes (like protein amount) are so difficult or un-intuitive for the crowd to evaluate, that the convergence to the final answer might be slow and thus require high budget [31]. Another disadvantage is that query attributes are handled separately and potential mutual information (for example between *dessert* and *calories*) which could be used to reduce the number of required questions, or to improve accuracy, is ignored.

A first solution to this problem was proposed in [27]. Instead of asking the crowd directly about the attributes mentioned in the query, it was suggested to also ask for the value of other (usually simpler) related attributes, and then derive the value of the query attributes from the answers. For example, to estimate the amount of protein in a certain dish one may ask what quantities of high protein ingredients (such as meat, dairy, eggs, nuts and soy) does it contains. In this solution, queries are processed in two steps: (1) An *offline* preprocessing phase that, given a query, determines which object's attributes should be asked about, how many users should estimate each of those attributes and how the obtained values should be assembled together, and (2) an *online* query evaluation phase, where each object in the database is processed using the scheme derived in the offline step. For example, consider a query about the protein

[1]In common crowdsourcing platforms, crowd questions have some (small) monetary cost, and thus the number of questions per object is typically bounded by the allocated budget

10.5441/002/edbt.2015.36

amount in recipes in CC. Assuming a budget of 20 questions per recipe, the preprocessing phase may derive a formula of the form: $0.5 protein\_amount^{(10)} + 0.13 grams\_of\_meat^{(3)} + 0.15 grams\_of\_dairy^{(4)} + 3 number\_of\_eggs^{(3)}$. In the formula, $attribute\_name^{(n)}$ denotes the average value of n users answers when asked about the given attribute. The formula indicates that rather than using the full budget to ask users directly about protein amount, a better estimation would be obtained by asking only 10 crowd members, then averaging the derived value with a linear combination of estimated values for three other related attributes - *grams_of_meat*, *grams_of_dairy*, *number_of_eggs* (computed by asking 3, 4 and 3 crowd members about them, respectively). We will explain later how such formulas are derived.

While this approach is shown to provide results superior to those obtained by asking the same overall number of questions only about the query attributes [27], a great obstacle is that it requires the use a *domain expert* that provides the list of related attributes for each query. This use of experts-in-the-loop limits the scalability of the approach and its applicability to fully-automated crowd-only platforms. In contrast, in the present paper, we provide a solution which is *entirely crowd-based*. Our goal is to replace the domain-expert by the Wisdom of Crowds, asking users to assist in "dismantling" difficult attributes and identifying those related attributes that can assist in query evaluation. Note that even harder related attributes may improve results they force different ways for estimations, which is one of the foundations of the wisdom of crowds principle.

As we will show, a successful solution will need to address two main challenges. The first is determining which object attributes should we best ask the crowd to dismantle. We show that it is sometimes beneficial to not only dismantle the query attributes themselves, but rather to continue dismantling newly discovered attributes. We provide a fine hypothetical analysis to estimate the potential benefit (and cost) of dismantling each of the so-far-discovered attributes and thereby determining which questions to ask the crowd. The second related challenge that we address is budget management. Given a budget (e.g., number of questions that can be asked to the crowd) for the offline preprocessing step, we need to use it both for dismantling the query attributes, as well as for obtaining some statistics about them (e.g., the distribution of user answers, the correlations between attributes value, etc.). Such statistics are required to estimate properties like "difficulty" and "contribution" of each attribute in order to decide how many crowd members should be asked about each attribute and how the answers should be assembled together. Our algorithm provides a careful analysis that allows to balance the budget between these two complimentary tasks.

We present in this article the following contributions:

1. We propose a simple and generic model for modeling a database of objects with infinite unknown attribute names and values, the type of questions that can be posed to the crowd and the characteristics of those answers.

2. Given an *online per-object budget* and an *offline preprocessing budget*, we use the model to present an algorithm that ideally uses the offline budget for deriving linear formulas (like the one illustrated above) that best exploit the online budget for deriving the values of query attributes. Our algorithm consists of five inter-related components for which we explain what type of information is required and what crowd questions may be used to obtain it. We provide a generic black-box description for each component (which allows to plug-in different implementations) and propose a concrete implementation.

3. Since the success of our framework depends on the discovery of relevant attributes, we focus our attention on this problem. We formally show how the potential gain (and cost) of each possible attribute dismantling question can be estimated and how this estimation can then be used to design an iterative algorithm that optimally chooses which crowd questions should be asked at each point. The estimation is based on a careful analysis of the already gathered information as well as on predictions about the potential effect of each question on the following algorithm components.

4. Of particular challenge are queries with more than one attribute. There is a fine tradeoff here between the gain that one may obtain by discovering underlying correlations between attributes, and the cost (in terms of crowd questions) required for such discovery. Our algorithm uses a fine analysis of the current data to predict potential contributions and to balance the two.

5. Finally, we present a thorough experimental analysis of our approach over two real-life data sets as well as synthetic data. We examine the various parts of our algorithm and its performance as a whole. We compare our algorithm to several existing/alternative approaches, showing that it consistently outperforms them in achieving lower average error for the same budget. Our experiments also demonstrate the necessity of different parts of the algorithm for accurate attribute dismantling, which later translates to accurate attribute value estimation.

The paper is organized as follows. The model and all relevant notations are presented in Section 2. To simplify the presentation we first consider in Section 3 the case where the query contains a single attribute. Queries with multiple attributes are then considered in Section 4. Our experimental study is presented in Section 5. We discuss related work in Section 6 and conclude in Section 7.

2. PRELIMINARIES

We start by describing our model and notations, then formally define the problem that we study.

Objects, attributes and queries. Our data set consists of a set of objects. We use $\mathcal{O}$ to denote the possibly infinite domain of objects, and o, o_i to denote an individual object in $\mathcal{O}$. In our running example, $\mathcal{O}$ is the set of all food recipes. An object may have attributes. We use $\mathcal{A}$ to denote the domain of attribute names and a,a_i to denote an individual attribute name in $\mathcal{A}$. We focus here on numerical attributes. Boolean attributes may be viewed here as numerical attributes with a value between 0 and 1, whereas multi-value attributes can be modeled by one such boolean attribute per value. In our running example, $\mathcal{A}$ includes all possible recipe properties such as *time_to_prepare*, *is_soup*, *is_tasty*, *protein_amount*, *is_brown*, *number_of_eggs*, etc.

For an object $o \in \mathcal{O}$, an attribute name $a \in \mathcal{A}$, a set of objects $O \subset \mathcal{O}$ and a set of attribute names $A \subset \mathcal{A}$, we use the following notations: (1) $o.a$ denotes the value of the attribute a of the object o, (2) $o.a^{(*)}$ denotes an estimation of that value (3) $o.A \equiv \{o.a \mid a \in A\}$ denotes the set of values for attributes in A of object o, (4) $O.a \equiv \{o.a \mid o \in O\}$ is the set of values of attribute a of the objects in O. We will sometimes abuse notations and consider these sets as *random variables* over objects in O (to be explained later). Finally (5) $D_{O\times A}$ denotes a data table with rows corresponding to objects in O and columns to attributes in A, along with some representation for each object.

Given a *query* Q over some data table D, we define $\mathcal{A}(Q)$ as the set of Q's query attributes. W.l.o.g one may think of Q as an SQL query and of $\mathcal{A}(Q)$ as the set of attribute names appearing in Q. In our running example Q might be, for instance, *select number_of_calories, protein_amount from CC where dessert=true.* In this example, $\mathcal{A}(Q) = \{$*is_dessert*, *number_of_calories*, *protein_amount*$\}$. As some attributes (and their values) may be missing from the data table D, we will need to learn them from the crowd.

Crowd questions. Crowd workers may be asked four types of questions:

Attribute Value Questions (for brevity, value questions) - Here a crowd member is asked to provide an estimation of the value of an *o.a*. An example of a value question in our running example is showing a worker a recipe and asking her for the value of *number_of_eggs*. For an object $o \in \mathcal{O}$ and an attribute $a \in \mathcal{A}$, $o.a^{(1)}$ denotes the random variable representing the estimation of one random worker's when asked about *o.a*. We use the $^{(1)}$ notation also for estimations of groups of values (for example, $O.A^{(1)} \equiv \{o.a^{(1)} \mid o \in O, a \in A\}$).

Attribute Dismantling Questions (for brevity, dismantling questions) - A crowd member is given here an attribute's name and requested to give another attribute's name that may provide some information about the value of the former. We assume (as later confirmed in our experiments) that workers are more likely to provide attributes that are correlative with the attribute in question. An example for a dismantling question may be *which recipe's attribute may help estimate its number_of_calories.* An answer may be *is_dietetic*. For simplicity we assume that answers that refer to the same property (like *large, big, grand*) can be reasonably identified and merged to a single representative. This may be done, e.g., using a common thesaurus/NLP tools. (We will show however that our technique can work even without this).

Dismantling Verification Questions (for brevity, verification questions) - Here we use crowd workers to verify that a previously suggested attribute a_i may indeed help in estimating the value of another attribute a_j. An example for a verification question is *does knowing if a dish is_black may help in determining its number_of_calories.* The likely crowd answer here is *No*.

Example Questions - Here workers are given some attributes' names and are asked to provide an example of an object $o \in \mathcal{O}$ along with its values for the attributes. An example for such a question is asking a user to upload a recipe along with its calorie value. For simplicity we will assume below that the given value is the correct one (otherwise the it can be estimated via value questions).

For all tasks we assume workers are independent and that spam filters are employed to avoid malicious workers.

Other notations. We further adopt and use some common notations. From statistics we use $\mathbb{E}_X[f(x)]$ for expectation, $\mathrm{Var}_X[f(x)]$ for variance, $\sigma_X(f(x))$ for standard deviation, $\mathrm{Cov}_{X,Y}(f(X), g(Y))$ for covariance and $\rho_{X,Y}(f(x), g(y))$ for correlation. The lower indexes specify the random variables and we omit them when they are clear from context. From algebra, we use M^T to denote a matrix M's transpose, M^{-1} to denote M's inverse and $\mathrm{Diag}(f(i))$ for diagonal matrix where the i'th value of the diagonal is equal to $f(i)$.

Problem definition. A user allocates a *per-object budget* B_{obj} which is the number of value questions that can be asked on a given object, in the online query evaluation phase, for estimating the value of the query attributes. To determine how to best use this budget, the user also allocates a *preprocessing budget* B_{prc} [2]. An (offline) preprocessing phase uses this to gather some information from the crowd (using the type of questions described above) and consequently derive a set of formulas of the form $o.a^{(*)} = \sum_{\mathcal{A}} l_a(a_i) o.a_i^{(b(a_i))}$ for each $a \in \mathcal{A}(Q)$, which determine how objects should be processed in the online query evaluation phase. The semantics of such formula is to first ask the crowd $b(a_i)$ value questions about each attribute $o.a_i$, then calculate the average answer for each $o.a_i$ (denoted $o.a_i^{(b(a_i))}$) and finally calculate an estimation for *o.a* using a *linear regression*[12] with predictors $l_a(a_i)$. An example of such a formula, for the attribute *protein_amount*, was given in the Introduction.

The function b in the formulas determines how many questions (if any) will be asked about each object's attribute, and intuitively reflects the "difficulty" of each attribute. Since the total value questions per object need to obey the B_{obj} budget constraint, b must satisfy $\sum_{a\in\mathcal{A}} b(a) \leq B_{\mathrm{obj}}$. We call such function b a *budget distribution* of size B_{obj}.

For a given budget distribution function b and a linear regression formula l, we define an error in the estimation of a single attribute's value as $Er(o.a^{(*)} \mid b, l) = (o.a - \sum_{\mathcal{A}} l_a(a_i) o.a_i^{(b(a_i))})^2$. We then define the error of an attribute estimation as the mean square error over all objects $Er(\mathcal{O}.a^{(*)} \mid b, l) = \mathbb{E}_{\mathcal{O}}[Er(o.a^{(*)} \mid b, l)]$ and the query error as $Er(Q(D)^{(*)} \mid b, l) = \sum_{a\in\mathcal{A}(Q)} Er(\mathcal{O}.a^{(*)} \mid b, l)$. Note that for simplification we assume the errors of all attributes to be of equal importance. All our results also apply to a weighted error definition, as discussed later. Our goal here will be to minimize the query error $Er(Q(D)^{(*)})$. Namely, to find b and l which minimize it, and to do this using at most a budget B_{prc}.

3. OUR SOLUTION

We start by presenting a high-level informal description of our algorithm, what data do we collect and what is that data used for. Next, we provide a detailed formal description of the functionality of the different components, as well as references to existing solutions for some of them. We then return to the components that are in the heart of our contribution and provide concrete novel solutions for them.

To simplify the presentation we will assume below that the query contains only a single attribute, that we call the

[2]W.l.o.g. it is assumed that $B_{\mathrm{prc}} >> B_{\mathrm{obj}}$

target attribute, namely $\mathcal{A}(Q) = \{a_t\}$. We consider the general case afterwards. We also assume that no attributes are initially available for the objects in the queried data table. For instance, in our running example this means that we are only given the recipes and no explicit attributes for them. The algorithm can be naturally extended to the general setting.

```
  Data: $Q, B_{\text{obj}}, B_{\text{prc}}$
1 $E_B \leftarrow$GetExamples$(N_1, k)$;
2 while $CollectingAttributesCondition = True$ do
3     $a \leftarrow$GetNextAttribute$(A, S, B_{\text{obj}})$;
4     $A \leftarrow A \cup a$;
5     $S \leftarrow$UpdateStatistics$(S, a, E_B)$;
6 $b \leftarrow$FindBudgetDistribution$(S)$;
7 $E_L \leftarrow$GetExamples$(N_2, b)$;
8 $l \leftarrow$FindRegression$(b, E_L)$;
9 return $l, b$
```

Algorithm 1: The base case solution

Algorithm 1 depicts a general description of our solution. Table 1 shows the different information items being collected throughout its execution. It contains objects (first column), true values of objects' attributes (second column), and sets of workers' answers to value questions (denoted $\{o_i.a_j^{(1)}\}_1^n$ in all other columns where n is the answers set size). We use this table to illustrate *what* is done by Algorithm 1. We later explain *how* exactly this is done (what crowd tasks are involved, etc.). Note that some notations in Table 1 may not yet be clear at this point, but will be explained later.

At the beginning, the only information available is the name of the query attribute (Table 1's "True Values for $\mathcal{A}(Q)$" header). We first collect a set of example objects $E_B = \{e_1, \ldots, e_{N_1}\} \in \mathcal{O}$ along with their true value for a_t. Those objects and values are shown in Table 1a. We then iteratively add new attribute columns in Table 1a by dismantling existing attributes, thereby discovering new attribute names (the "Value Questions Answers for A_{final}" headers) and then obtaining values for them from workers. During this iterative process, we also use the crowd answers to calculate some statistics (not shown in the table) on the discovered attributes, which we use for deciding which attributes need to be dismantled next. When this collection process ends (we will explain later how this is determined), we use the statistics again to calculate a budget distribution b. Finally, to compute the linear regression l we collect a second set of examples E_L, along with their value for a_t (Table 1b's Objects and True Value columns). We use b to collect crowd answers for the remaining attributes, then use all the gathered information to learn the linear regression l. Now that we derived both b and l, the preprocessing phase ends.

Later, in the query evaluation phase, b and l are used to collect estimations about the objects in the queried data table D (the Answers in 1c) and use it to calculate and return $Q(D)$ (the "True Values" column in 1c).

3.1 The Algorithm Components

Algorithm 1 consists of five logical components: finding relevant attributes (lines 3-4), collecting statistics about them (lines 1 and 5), calculating a budget distribution (line 6), learning a linear regression (lines 7-8) and managing the preprocessing budget (line 2). We discuss them next.

objects	True Values for $\mathcal{A}(Q)$	Value Questions Answers for A_{final}			
	a_1	a_1	a_2	$\cdots$	a_l
e_1	$e_1.a_1$	$\{e_1.a_1^{(1)}\}_1^k$	$\{e_1.a_2^{(1)}\}_1^k$	$\ldots$	$\{e_1.a_l^{(1)}\}_1^k$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
e_{N_1}	$e_{N_1}.a_1$	$\{e_{N_1}.a_1^{(1)}\}_1^k$	$\{e_{N_1}.a_2^{(1)}\}_1^k$	$\ldots$	$\{e_{N_1}.a_l^{(1)}\}_1^k$

(a) Data used to calculate b

objects	True Values for $\mathcal{A}(Q)$	Value Questions Answers for A_{final}			
	a_1	a_1	a_2	$\cdots$	a_l
e_1	$e_1.a_1$	$\{e_1.a_1^{(1)}\}_1^{b(a_1)}$	$\{e_1.a_2^{(1)}\}_1^{b(a_2)}$	$\ldots$	$\{e_1.a_l^{(1)}\}_1^{b(a_l)}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
e_{N_2}	$e_{N_2}.a_1$	$\{e_{N_2}.a_1^{(1)}\}_1^{b(a_1)}$	$\{e_{N_2}.a_2^{(1)}\}_1^{b(a_2)}$	$\ldots$	$\{e_{N_2}.a_l^{(1)}\}_1^{b(a_l)}$

(b) Data used to learn l

objects	True Values for $\mathcal{A}(Q)$	Value Questions Answers for A_{final}			
	a_1	a_1	a_2	$\cdots$	a_l
o_1	?	$\{o_1.a_1^{(1)}\}_1^{b(a_1)}$	$\{o_1.a_2^{(1)}\}_1^{b(a_2)}$	$\ldots$	$\{o_1.a_l^{(1)}\}_1^{b(a_l)}$
o_2	?	$\{o_2.a_1^{(1)}\}_1^{b(a_1)}$	$\{o_2.a_2^{(1)}\}_1^{b(a_2)}$	$\ldots$	$\{o_2.a_l^{(1)}\}_1^{b(a_l)}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

(c) Data used in the online phase

Table 1: Data collected during the algorithm

Finding Attributes. We identify here a set of attributes that may assist in estimating the value of the query attribute. This is done using dismantling questions, followed by corresponding verification questions. A key observation here is that it is sometimes beneficial to not only ask users to dismantle the query attribute itself, but rather to continue dismantling newly discovered attributes. Indeed, since the human mind is associative, asking diverse questions is important for better learning a domain [7]. For example, in our CC example, when asking a user to dismantle *protein_amount*, we may get the attribute *meat_content* as an answer, but the distinction between *red_meat* and *white_meat* (which have different protein amounts) may be only obtained when asking users to dismantle *meat_content*.

We denote by A_m the set of known related attributes after m iterations (and respectively $A_0 = \mathcal{A}(Q)$ and A_{final} is the final subset). We denote by $A_{m+1|a_j} = A_m \cup ans_j$ the random variable representing this set, assuming the next dismantling question is for attribute a_j. Our goal will be to choose a_j such that $A_{m|a_j}$ allows minimal error. More formally, we wish to find

$$\operatorname{argmax}_{a_j} \mathbb{E}_{A_{m|a_j}=A}\Big[\min_{\substack{l,b \\ \forall a \notin A \;\; b(a)=0}} Er(Q|b,l)\Big] \qquad (1)$$

As this choice obviously depends on how the budget distribution b and the linear regression l are selected, we leave the solution of expression 1 for section 3.2.1. For now we only note that after asking the selected dismantling question and getting a new attribute name for an answer, we use verification questions to ensure that the obtained new attribute name is indeed a relevant one. Here we use standard algorithms such as [25] to determine the required number of questions for making a decision.

Since a dismantling question in our setting is always followed by corresponding verifications questions, from here on whenever we use the term dismantling question we also refer to its following verification questions.

Collecting Statistics. As mentioned, we need to collect some information about A_m - the set of known related attributes after m iterations - and the way workers answer

			$\mathcal{A}(Q)$	A_m			
			a_1	a_1	a_2	$\cdots$	a_l
A_m	a_1	$S_c[1]$	$S_o[1]$	$S_a[1,1]$	$S_a[1,2]$	$\cdots$	$S_a[1,l]$
	a_2	$S_c[2]$	$S_o[2]$	$S_a[2,1]$	$S_a[2,2]$	$\cdots$	$S_a[2,l]$
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
	a_l	$S_c[l]$	$S_o[l]$	$S_a[l,1]$	$S_a[l,2]$	$\cdots$	$S_a[l,l]$

Table 2: Statistics calculated during the algorithm

them. Our main tool for finding those statistics is gathering samples of the crowd responses and analyzing them. We do so by asking value questions about a set of example objects collected using example questions. Formally, our goal is to find an accurate estimation for the trio $S_A = (S_{oA}, S_{aA}, S_{cA})$ (or just S when the index is clear from context), depicted in Table 2 and defined as follow (for reasons that will be clear later).

S_c - Statistics about agreement among crowd workers. More precisely, a vector of the average variances of workers answers to value questions. Formally, S_{cA} is a vector of size $|A|$ where $S_{cA}[a] = \mathbb{E}_{\mathcal{O}}[\text{Var}[o.a^{(1)}]])$. For instance, in our example we can expect $S_c[\textit{healthy}] > S_c[\textit{tomato}]$ as it is easier to identify if a recipe contains a tomato. This is the second column of Table 2.

S_o - Statistics about how informative are the attributes. More precisely, this is the covariance vector between workers answers to the different attribute and the query attribute. Formally, S_{oA} is a vector of size $|A|$ and $S_{oA}[a] = |\text{Cov}_{\mathcal{O}}(o.a^{(1)}, o.a_t)|$. For example, if $a_t = \textit{dessert}$, we can expect $\frac{S_o[\textit{sweet}]}{\sigma(\textit{sweet})} > \frac{S_c[\textit{cheese}]}{\sigma(\textit{cheese})}$ as most desserts are sweet (and most non-desserts are not) but cheese can be easily found both in desserts and in non-desserts. This is the third column of Table 2.

S_a - Statistics about how much distinctive are attributes (in comparison to the other attributes). More precisely, this is the covariance matrix over crowd's answers to different attributes. Formally, S_{aA} is a matrix of size $|A| \times |A|$ and $S_{aA}[a_i, a_j] = |\text{Cov}_{\mathcal{O}}(o.a_i^{(1)}, o.a_j^{(1)})|$. For example, we can expect $\frac{S_a[\text{Spicy, Sugar}]}{\sigma(\text{Spicy})\sigma(\text{Sugar})} > \frac{S_a[\text{Easy to make,Sugar}]}{\sigma(\text{Easy to make})\sigma(\text{Sugar})}$ as sugar usually indicates a non-spicy food but does not imply anything about the complexity of the recipe. This is the fourth column of Table 2.

Our goal is to get relatively good estimations of these measures for a low budget. We describe how this is done in section 3.2.2.

Calculating a budget distribution . Once the relevant attributes are identified and the relevant statistics are calculated we run a strictly computational algorithm to find b. As it was shown in [27], when applying the best linear regression to some table $D^{(b)}_{O\times A}$ (a notation that means $D^{(b)}_{O\times A}[o, a] = o.a^{(b(a))}$), the error is $\mathbb{E}[E.a_t]^2 - {S_o}^T(S_a + \textit{Diag}(\frac{S_c(a)}{b(a)}))^{-1}S_o$. The first element is independent of b, so we have that the best budget distribution is

$$\underset{b}{\text{argmax}}\, {S_o}^T(S_a + \textit{Diag}(\frac{S_c(a)}{b(a)}))^{-1}S_o \tag{2}$$

[27] also showed that finding this optimal b is Np-hard in B_{obj} and therefore an approximating algorithm is appropriate. They provide such algorithm, which is a variation of the well known greedy forward selection. We use this algorithm as the *FindQuestionsDistribution* method in Algorithm 1.

Learning a Linear Regression. The last part of the algorithm is deciding on a linear regression l. Since we can not find the overall best linear regression, we minimize the error over some training set representing the online phase data (meaning that the estimation of each attribute a is done according to $b(a)$). We get this set by using example questions (getting object and target values) and value questions (getting attribute values). To reduce costs we re-use previously collected data. When collecting objects we skip the first N_1 example questions, and when collecting estimations we only ask $b(a) - k$ value questions for each $e.a.$ This is line 7 in Algorithm 1 and how we collect Table 1b's data.

Once such training set exists, further computations are applied to find a linear regression l that minimizes the error over it. The problem of finding a linear regression that minimizes the mean square error is a well studied problem [12] and there are many algorithms for it that we can just use. Specifically, we used a singular value decomposition (SVD [15]) algorithm, but since it is used as a black box other algorithms can also fit. This is line 8 in Algorithm 1.

Managing the Preprocessing Budget. To fully understand where and how budget is spent, one needs to first see the actual implementation presented next. We thus postpone this discussion to section 3.2.3.

3.2 Concrete Solutions

Finally, we can focus on our implementations. We wish to remind that although they are described separately, all the components are in fact intertwined.

3.2.1 Finding Attributes

Recall that our objective is to solve expression 1. Knowing now, that the error behaves like expression 2 we can state a more specific objective - finding

$$\underset{a_j}{\text{argmax}}\,\underset{b}{\text{argmax}}\, S_{oA_{m|a_j}}^T (S_{aA_{m|a_j}} + \textit{Diag}(\frac{S_{cA_{m|a_j}}(a)}{b(a)}))^{-1} S_{oA_{m|a_j}} \tag{3}$$

As an exact solution can only be made after asking all questions and calculating all $S_{A_{m|a_j}}$, we use the current statistics $S_{A_{m-1}}$ and estimate the next statistics $S_{A_{m|a_j}}$, for every a_j. We also calculate the probability of it remaining the same as only a first seen answer will effet it. We then use those estimations and solve expression 3. Described here are general schemes of the estimations. Full calculations can be found in the our paper[22].

$\mathbf{Pr(new} \mid \boldsymbol{a_j})$ - We need to estimate the probability to get a new answer. We do so by assuming it depends only on the number of questions asked so far and then using a simple Bernoulli-Bayesian model with the number of questions asked about a_j so far (n_j). The results are

$$\Pr(\text{new} \mid a_j) = \frac{n_j + 1}{n_j^2 + 3n_j + 2} \tag{4}$$

$\boldsymbol{S_{oA_{m|a_j}}}$ - We need to estimate the covariance of the next answer and the target ($S_o[ans_j]$). By definition of correlation we have $S_o[ans_j] = \frac{\rho(a_t, ans_j)}{\rho(a_t, a_j)} \frac{\sigma(ans_j)}{\sigma(a_j)} S_o[a_j]$. $\sigma(a_j)$ and $S_o[a_j]$ are known. $\sigma(ans_j)$ is assumed to be independent with a_j and can therefore be ignored. That leaves only the correlations' ratio. We previously assumed ans_j is highly correlated to a_j, we now approximate this as $\mathbb{E}[\rho(a_j, ans_j)] \approx 0.5$, which translates to $\frac{\rho(a_t, ans_j)}{\rho(a_t, a_j)} \approx 0.5$. We address this approximation later. This results in

$$S_{oA_{m|a_j}}[a] \approx \begin{cases} \frac{0.5}{\sigma(a_j)} S_{oA_{m-1}}[a_j] & a = ans_j \\ S_{oA_{m-1}}[a] & \text{otherwise} \end{cases} \quad (5)$$

$\boldsymbol{S_{cA_{m|a_j}}}$ - We need to estimate the variance of the next answer ($S_c[ans_j]$). Since there is no reason for it to change over different dismantling questions we can use the same distribution for every j. Because *FindRegression* is not analytic (as we will see later), instead of measuring an exact distribution (which will make it impossible to calculate) we take an 'optimism in the face of uncertainty' approach [20] and assume a very low constant value for it ($\forall j\ S_c(ans_j) \approx 0$). We then get

$$S_{cA_{m|a_j}}[a] \approx \begin{cases} 0 & a = ans_j \\ S_{cA_{m-1}}[a] & \text{otherwise} \end{cases} \quad (6)$$

$\boldsymbol{S_{aA_{m|a_j}}}$ - We need to estimate the covariances between the new answer and the previously discovered attributes ($S_a[a_i, ans_j],\ a_i \in A_{m-1}$). Again, since there is no reason for this to change over different dismantling questions we can just take the same distribution for every j. For similar reasons (calculations practicality), we again take the 'optimism in the face of uncertainty' approach. We (wrongfully) assume no correlation between the new and the existing attributes. This assumption cannot be taken for $S_a[ans_j, ans_j]$, but this factor cancelled anyway in the $Er(Q)$ calculation. We then get

$$S_{aA_{m|a_j}}[a_u, a_v] \approx \begin{cases} 1 & a_u = a_v = ans_j \\ 0 & ans_j \in \{a_u, a_v\} \\ S_{aA_{m-1}}[a_u, a_v] & \text{otherwise} \end{cases} \quad (7)$$

By putting results 4, 5, 6 and 7 into expression 3 we get that the best next dismantling question is

$$\underset{a_j}{\operatorname{argmax}} \Pr(\text{new} \mid a_j)[G(a_j) - L(A_{m-1}, B_{\text{obj}}, 1)] \quad (8)$$

where $\Pr(\text{new} \mid a_j)$ was described before, $G(a_j) = \frac{0.25 S_o[a_j]^2}{\sigma(a_j)^2}$ and $L(A, u, v) = \max_{b \text{ of size } u} S_{oA}(S_{aA} + Diag(\frac{S_{cA}}{b}))^{-1} S_{oA} - \max_{b \text{ of size } u-v} S_{oA}(S_{aA} + Diag(\frac{S_{cA}}{b}))^{-1} S_{oA}$.
Intuitively, when adding a new attribute, some of the record budget moves from the old attributes to the new one. G measures the gain from the new attribute and L measures the loss caused by decreasing budget from the old attributes. The reasons for those being the only changes are the low correlation assumptions we took while estimating $S_{A_{m|a_j}}$.

As all of those values can be calculated, this concludes the *GetNextAttribute* method in Algorithm 1. This is also how we get each "Value Questions Answers" header in Table 1. It is easy to see that one dismantling question at the end of each iteration is the only crowd task.

3.2.2 Collecting Statistics

As explained, our goal in this part is to find a good approximation for (S_{oA}, S_{aA}, S_{cA}) while using a minimal budget. Following our iterative process for finding attributes, we build S in an inductive way. Namely, for each new attribute we calculate S_{A_m} based on $S_{A_{m-1}}$ and questions about the new attribute.

For our current simplified case, there exists an approximation method in [27] that have proven itself before and that we can easily adapt. When we later discuss the general case, we will return to this part and refine the calculation. The ideas we take from [27] are to estimate $S_{A_{\text{final}}}$ (which is defined over $\mathcal{O}$) by calculating it over example set E_B and then, for each object, estimating the behavior of $o.a^{(1)}$ based on k sample answers (for a very small k). We use those ideas in an inductive way

A_{-1} - We leave S_c,S_a,S_o empty but collect a set of examples E_B with target a_t value by asking N_1 example questions (N_1 is a parameter studied in [27]). This is line 1 in Algorithm 1 and during it we collect Table 1a's objects and true values.

A_m - For the new attribute a we ask k values questions about $e.a$ for every $e \in E_B$. We then update S by keeping all previous values and adding (1) $S_o[a] = \mathbb{E}_{E_B}[e.a^{(k)} \cdot e.a_t]$, (2) $S_a[a, a_i] = S_a[a_i, a] = \mathbb{E}_{E_B}[e.a^{(k)} \cdot e.a_i^{(k)}]$ for every $a_i \in A_m$ and (3) $S_c[a] = \mathbb{E}_{E_B}[\mathit{VarEst}_k(e.a^{(1)})]$. This is the *UpdateStatistics* method in algorithm 1. During each step we get a sub-column in the answers column of Table 1a, a row in Table 2 and a sub-column in the right column of Table 2.

It should be easy to see how this algorithm is compliant with the ideas mentioned above. It should also be easy to see that during this part we use the crowd for N_1 example questions and $kN_1|A_{\text{final}}|$ value questions.

3.2.3 Management of the Preprocessing Budget

In our algorithm the preprocessing budget is used for

- Finding A_{final} by asking attributes and attributes verification questions. This costs n dismantle questions where n is the number of dismantling questions we choose to ask.
- Calculating the statistics $S_{A_{\text{final}}}$ by asking example and value questions. This costs N_1 example questions and $kN_1|A_{\text{final}}|$ value questions, where $|A_{\text{final}}|$ depends on n.
- Collecting a training set of N_2 samples for l's learning. This costs $(N_2{-}N_1)$ example questions and $(N_2{-}N_1)B_{\text{obj}} + N_1(B_{\text{obj}} - \sum_{\mathcal{A}} \min\{b(a), k\})$ value questions.

As N_1 and k are external parameters, the only variables are n and N_2. Therefore, the only open question is when to stop asking dismantling questions (line 2 in algorithm 1). We solve this tradeoff by applying a common simple linear lower bound for N_2 as a function of b ([16]). Note that as our only tradeoff is n vs. N_2, this mechanism is also appropriate when considering different costs for different crowd tasks.
As to the case of different cost that may apply for different questions of the same type (for example, numeric vs. binary), the appropriate coefficients need to be added. In this case, we also follow [27] idea and, during all components, divide each attribute's contribution by its cost.

Remarks. We conclude this section by commenting on the correctness and complexity of the algorithm. First, it is easy to see that the algorithm operates within the preprocessing budget B_{prc}. Second, it is also easy to see that the algorithm running time is polynomial with respect to the two budgets B_{prc} and B_{obj}.

4. EXTENDED SOLUTION

We focused so far on the simplified case where the query has a single attribute. We next consider the general case of multiple query attributes.

A naive solution is to equally split the online and offline budgets between the query attribute and solve the problem for each one separately. This however ignores possible correlation between the query attributes and their components. For example, consider a query with two attributes $\mathcal{A}(Q) = \{\textit{calories, is\_dessert}\}$. It is easy to see that many related attributes (e.g., *sugar, fat,...*) are good indicators for both target attributes, and budget would be saved if we reuse values. To address this we consider all the query attributes together, extending Algorithm 1 to handle multiple target attributes. We first present below a simple extension, then discuss its shortcomings, and then generalize it to overcome them. Our first extension generalized the algorithm components as follows.

GetExamples - Instead of asking the crowd for examples with one value for the single query attribute we now ask for examples with multiple attribute values - one per query attribute. (We later discuss what to do if users cannot provide all these values simultaneously.)

GetNextAttribute - Expression 8 is refined to consider all the attributes:

$$\underset{a_j}{\operatorname{argmax}} \sum_{a_t} \Pr(\text{new} \mid a_j)[G(a_t, a_j) - L(a_t, A_{m-1}, B_{\text{obj}}, 1)] \tag{9}$$

UpdateStatistic - Instead of updating the S_o statistics of a_{new} for a single query attribute, we now need to update $S_{oa_t}[a_{\text{new}}]$ for every query attribute $a_t \in \mathcal{A}(Q)$. Note that we added here the a_t notation to S_o since there are now several query attributes. S_a and S_c remained the same since they are independent of Q.

FindQuestionsDistribution - Instead of equation 2 we now have a refined version

$$\underset{b}{\operatorname{argmax}} \sum_{a_t \in \mathcal{A}(Q)} S_{o(a_t,A)}^{T} (S_{aA} + Diag(\frac{S_c[a]}{b(a)})^{-1} S_{o(a_t,A)} \tag{10}$$

FindRegression - Since l is now a set of linear regressions, we need to run this method $|\mathcal{A}(Q)|$ times. Since all l_{a_t} are independent this yields an optimal solution.

Note that for *GetExamples* we assumed above that it is possible to ask workers for examples with several attributes values. This may be problematic in practice: If the number of query attributes is too large, workers may not be willing to make the effort of providing all of their values; It may also be the case that a single crowd member does not know the value of all attributes, even for their own examples. To overcome this, instead of using just one set of examples E_B with all query attributes, we will collect multiple sets of examples $E_{B_{a_t}}$, one for each query attribute (or a small subset thereof). In this case, the collected data in Table 1a is replaced by the one depicted in Table 3.

Looking at this table we can see that although the information can be used to derive b, it comes with an additional cost: It is easy to see that the amount of data

	True Values for $\mathcal{A}(Q)$		Value Questions Answers for A_{final}		
	a_1	a_2	a_1	$\cdots$	a_l
e_1	$e_1.a_1$	?	$\{e_1.a_1^{(1)}\}_1^k$	$\ldots$	$\{e_1.a_l^{(1)}\}_1^k$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
e_{N_1}	$e_{N_1}.a_1$	?	$\{e_{N_1}.a_1^{(1)}\}_1^k$	$\ldots$	$\{e_{N_1}.a_l^{(1)}\}_1^k$
e_{N_1+1}	?	$e_{N_1+1}.a_2$	$\{e_{N_1+1}.a_1^{(1)}\}_1^k$	$\ldots$	$\{e_{N_1+1}.a_l^{(1)}\}_1^k$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
e_{2N_1}	?	$e_{N_1+1}.a_2$	$\{e_{2N_1}.a_1^{(1)}\}_1^k$	$\ldots$	$\{e_{2N_1}.a_l^{(1)}\}_1^k$

Table 3: Data collected in the general case

we need to collect now depends both on $|A_{\text{final}}|$ and on $|\mathcal{A}(Q)|$. Therefore, if we want to allow A_{final} to grow with $\mathcal{A}(Q)$, our cost will grow quadratically. It is also easy to see that most of this growth is due to cost of redundant value questions. For example, consider $\mathcal{A}(Q) = \{$*is_dessert, number_of_calories,protein_amount, easy_to_make*$\}$. One can assume that although there is likely to be a correlation between *number_of_calories* and *is_dessert* this is not the case for *easy_to_make* and *protein_amount* so collecting statistics for all pairs is a waste of budget. To reduce this redundant overhead we take two steps. First, we choose carefully which data to collect (i.e., which $E_{B_{a_t}}.a$ value questions should we asked). Second, for pairs for which data had not been collected, we estimate $S_{oa_t}[a]$ based on the other collected data. We explain this next.

Collection. Our choice of which data to collect is based on the following observation. The two cases one wants to avoid are (1) missing a highly correlated attribute-target pair and (2) wasting budget on a poorly correlated attribute-target pair. Therefore, whenever we get a new attribute a_j we pair it with all query attributes a_t for which we have a reason to believe that $S_{oa_t}[a_j]$ is not negligible. In our heuristic, we define $S_{oa_t}[a_j]$ as negligible iff its value is less than a half of the maximal value $\max_{a \in \mathcal{A}(Q)} S_{oa}[a_j]$. Our estimation here for $S_{oa_t}[a_j]$ is done in the same way we described earlier in section 3.2.1. This results in the following rule - when asking a dismantling question about a_i and getting an answer a_j, ask value questions about $E_{B_{a_t}}.a_j$ iff $\rho(a_i, a_t) > 0.5 \max_{a \in \mathcal{A}(Q)} \rho(a_j, a)$.

Estimation. Finally, to estimate the missing S_o's values we use a graph model and define $G = (U, V, E)$ as a weighted bipartite graph with $U(G) = \mathcal{A}(Q)$ and $V(G) = A_m$. The idea is to make each edge's weight $w(a, a_t)$ represent the value of $S_{oa_t}[a]$ and then estimating missing edges by distances on a graph. Ideally, we would have defined $w(a, a_t) = S_{oa_t}[a]$. However, since S_o is not normalized and also not a distance function, this is impossible. To overcome this, we employ a method described in [29] and use angular distance as our weight function - $w(a_t, a_j) = \Gamma(O.a_t, O.a_j) = \arccos \frac{S_{oa_t}[a_j]}{\sigma(a_t)\sigma(a_j)}$. The idea behind angular distance is to consider an inner-product space where the vectors are random variables and the inner-product is covariance. This allows to prove that Γ is indeed a distance function that answer what we looked for. Using the fact that in the angular distance space $\Gamma_1 + \Gamma_2 = \arccos(\cos(\Gamma_1)\cos(\Gamma_2))$, we define our estimation of S_o as

$$S_{oa_t}[a_j] = \sigma(a_t) \cdot \sigma(a_j) \cdot \begin{cases} \cos(w(a_j, a_i)) & \text{edge exists} \\ \cos(\text{S.P}(a_t, a_j)) & \text{path exists} \\ 0 & \text{otherwise} \end{cases} \tag{11}$$

where S.P stands for (multiplication) shortest path.

Weighted query attributes. To conclude, note that in our discussion so far we assumed the errors of all attributes to be of equal weight. In practice some normalization may be required. For example, *is_healthy* is on a scale of $[0, 1]$ whereas *number_of_calories* may reach thousands. In this case, each $a_t \in \mathcal{A}(Q)$ is associated with a weight ω_t and our goal is to minimize $\sum_{a_t \in \mathcal{A}(Q)} \omega_t \mathbb{E}[(O.a_t^{(*)} - O.a_t)^2]$. When following the previous calculation this simply results in adding weights to expression 9.

5. EXPERIMENTS

We analyze our solution experimentally along through dimensions. We start with a general proof of concept - an examination of our algorithm as a whole. We then move to an analysis of its components, their necessity and their quality. We conclude with an analysis of how different assumptions and parameters can influence the results.

5.1 Experiments Settings and Datasets

We used three datasets - two with real life objects and real crowd answers, and one synthetic. Crowd answers from value and attribute questions were gathered through CrowdFlower[3] - a platform for presenting small tasks to crowds. The answers collected in initial experiments was recorded in a database and reused in following experiments, so that results of multiple runs/algorithms may be compared in equivalent settings. To compare our performance to [27] that used experts to obtain relevant attributes, we also added to our database the data collected in that work. For example questions, in order to have a true 'gold standard' (known target answers), we used our lab members as crowd.

We designed our crowd interface and payment following the guidelines in [13] and the work of [27]. Our crowd tasks consist of a set of value (resp. dismantling) questions that a crowd member needs to answer. We set the payment for binary value question to 0.1¢ and to 0.4¢ for general numeric values. For dismantling and example questions, that were not studied in [27], we set the payment to 1.5¢ per answer, following our preliminary experiments that showed this to be the minimal price that kept workers' feedback positive, and set the price of an example question to 5¢ as this is a relatively hard task. (We will show however in the sequel that the trends in our results are robust to changes in these numbers). As for other parameters we used the following: the number of value samples k used for estimating statistics when deriving budget distributions was 2, as this is the recommended number for the corresponding black-box that we used[27]. The number of examples N_1 was set to 200 to keep our costs low while still having many examples. For learning the linear regression, the examples number N_2 was set to $50+8*\#$attributes, a common practice in such tasks [16]. For attribute weights, unless otherwise stated, we gave each query attribute a weight in reverse proportion to its variance ($\omega_t = \frac{1}{Var(O.a_t)}$). This normalize all errors to a similar scale (standard deviations), so that no query attribute will be negligible. We will explicitly mention below where using other weights affects the results.

Human Pictures Data Set. In this set of experiments our objects are people and the only information available for them is their picture. The query attributes in the different experiments include *Weight, Height, Age, Bmi* (body mass index, defined as $\frac{\text{weight(kg)}}{\text{height(m)}^2}$) and *Attractiveness.* The objects O were taken from the publicly available Photographic Height/Weight Chart [4], where people post pictures of themselves announcing their own height and weight. We used reported value as the true values for *Height, Weight* and *Bmi.* For other target values, we used an average over many value question estimations.

Question	Answer	Frequency
Bmi	Weight	33%
	Height	33%
	Age	6%
	Attrctive	2%
Height	Age	22%
	Shoe Size	9%
	Taller Then You	7%
	Weight	6%
Age	Wrinkles	15%
	Gray Hair	10%
	Old	10%
	Children	3%
Attractive	Good Facial Features	17%
	Fat	6%
	Has Good Style	6%
	Works Out	1%

(a) Pictures Domain

Question	Answer	Frequency
Calories	Has Eggs	8%
	Low Calories	4%
	Dessert	2%
	Healthy	2%
Protein	Has Meat	13%
	Number of Eggs	4%
	High Protein	4%
	Vegetarian	2%
Healthy	Low Salt	8%
	Natural	8%
	Fat Amount	4%
	Bitter	4%
Easy To Make	Number of Ingredients	17%
	Fast	10%
	Tasty	5%
	Expensive	2%

(b) Recipes Domain

Table 4: Attribute dismantling questions and their answers

	S_c	$S_o\sigma(a_i)\sigma(a_j)$		$S_a\sigma(a_i)\sigma(a_j)$					
		Bmi	Age	Bmi	Weight	Heavy	Attractive	Works Out	Wrinkles
Bmi	30	0.88	0.63	1	0.94	0.86	0.48	0.4	0.26
Weight	189	0.86	0.7	0.94	1	0.82	0.53	0.39	0.28
Heavy	0.14	0.89	0.6	0.86	0.82	1	0.44	0.46	0.27
Attractive	0.13	0.45	0.44	0.48	0.53	0.44	1	0.32	0.28
Works Out	0.11	0.36	0.29	0.4	0.39	0.46	0.32	1	0.15
Wrinkles	0.16	0.25	0.52	0.26	0.28	0.27	0.28	0.15	1

(a) Pictures Domain

	S_c	$S_o\sigma(a_i)\sigma(a_j)$		$S_a\sigma(a_i)\sigma(a_j)$					
		Calories	Protein	Calories	Low Calorie	Desset	Healty	Vegetarian	Eggs
Calories	80707	0.41	0.34	1	0.2	0.07	0.15	0.18	0.03
Low Calorie	0.06	0.18	0.08	0.2	1	0.1	0.26	0.1	0.13
Desset	0.08	0.26	0.5	0.07	0.1	1	0.44	0.34	0.38
Healthy	0.2	0.02	0.16	0.15	0.26	0.44	1	0.06	0.27
Vegetarian	0.13	0.26	0.52	0.18	0.1	0.34	0.06	1	0.14
Eggs	0.05	0.11	0.26	0.03	0.13	0.38	0.27	0.14	1

(b) Recipes Domain

Table 5: Examples for statistics in the different domains

Examples of answers received when asking workers to dismantle various attributes are depicted in Table 4. The first column depicts the attribute to dismantle, the second column contains some related attributes suggested by the crowd, the last column shows the percentage of all answers that each attribute name was returned. Examples of statistics gathered for the attributes are depicted in Table 5 (this is a concrete example of Table 2, but unlike Table 2 it shows attributes correlation and not covariance to make things more intuitive for the reader).

Recipes Data Set. In this set of experiments our objects are recipes and the data available for them in the database is the recipe's name, picture and unstructured ingredients-list. The query attributes in the different experiments include *Proteins, Calories, Good_for_kids, Easy_to_make* and *Healthy.* The objects are the 500 most popular recipes in `allrecipes.com`[1] website, normalized to one serving. We used nutritious values found in this website as true values for the matching query attributes. For other query attributes we again used average value derived from multiple value questions. Here again, examples of some answers obtained for dismantling questions and statistics on the attributes are depicted in tables 4 and 5 respectively.

Synthetic Data. To neutralize our own subjectivity/belief w.r.t which object attributes are hard/easy, we also ran experiments on a synthetically generated domain. For this we automatically generated a set of objects and attributes (with some dependencies between them) and mocked crowd answers about them (in compliance with the assumptions on crowd's answers mentioned in the paper). The details of this process can be found in the full paper [22]. The experiment results are consistent with those for real-life data and are thus omitted here.

5.2 Proof of concept

We compared our algorithm, which we call *DisQ* (short for Dismantling queries), to existing practices. We use the following algorithms as baselines:

NaiveAverage - In this common approach, the online phase simply asks questions about the attributes in $\mathcal{A}(Q)$ and returns their average $o.a_t^{(B_{\text{obj}})}$. For $|\mathcal{A}(Q)| > 1$ we split the budget by the weights. This algorithm has no offline preprocessing phase.

SimpleDisQ - This is a simplified version of our algorithm, which captures the best that can be done today without using an expert. It runs similar to *DisQ*, but without the attribute dismantling phase.

We compared these two algorithms to our algorithm. We did so for all three data sets and for different query attributes and query sizes. We also tested with different preprocessing budgets B_{prc} and different per-object budgets B_{obj}. For B_{obj} we used the range of 0.4-10¢. The lower bound was set to match 1 numeric value-question. The upper bound was set as it is a fairly large amount and as most of the experiment graphs show stagnation after it. For B_{prc} we used the range of \$10-35. We have taken those values since the graphs stagnate outside those boundaries. For each value we executed 30 experiments and took the average result. Note that although we took the average, all observations are true in general as most results are very close to the average.

Varying $\boldsymbol{B}_{\text{prc}}$. Figure 1a shows results for a query with $\mathcal{A}(Q) = \{Bmi\}$ (using the pictures data set) for varying preprocessing budgets. We will show more results later. We start with an example where $|\mathcal{A}(Q)| = 1$ to isolate different effects. We fixed B_{obj} to 4¢ and used different B_{obj} values. We used $B_{\text{obj}} = 4$¢ as it is over the graph's knee (as we will see later). Note that since *NaiveAverage* does not involve learning and since the number of examples in *SimpleDisQ* is always N_1 (since A_{final} is very small), *DisQ* is the only algorithm that changes with B_{prc}. One can easily see that for every B_{prc} value our algorithm has the lowest average error. The difference is especially significant for large B_{prc} values as for those ranges A_{final} is bigger. We can also see that the improvement is slowly stagnating which is the expected result if the "important" attributes are found quickly.
To ilustrate how an algorithm's output looks like, we provide here an example for one of the dismantles when $B_{\text{prc}} = \$25$. $Bmi^{(*)} = 0.6Bmi^{(5)} + 11.9Heavy^{(10)} + 0.4Works\_Out^{(1)} + 0.2Age^{(1)} - 2.7Attractive^{(3)} - 0.2Tall^{(2)} + 10.6$.

Varying $\boldsymbol{B}_{\text{obj}}$. We continue with the same *Bmi* example but now considering varying online per-object budget. Figure 1d show the errors for this case. We used $B_{\text{prc}} = \$30$ as it is

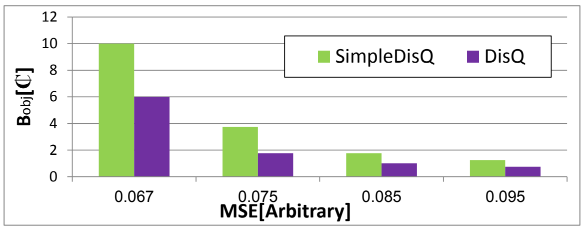
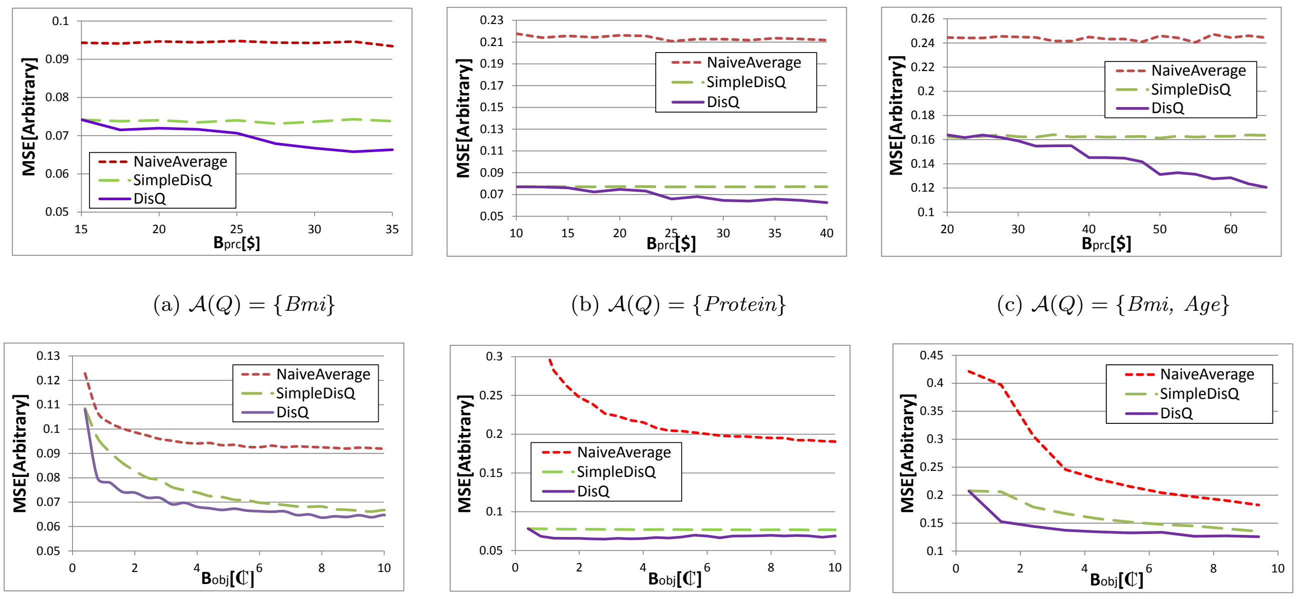


Figure 2: Necessary B_{obj} for achieving target errors

again over the graphs' knee, but similar behavior is shown for other values. First, note that all algorithms improve as B_{obj} increases and that this improvement is slowly decaying. This is what one could expect as a bigger B_{obj} means a bigger crowd (and should therefore mean better accuracy) and since it is known that every additional worker has declining marginal utility. Second, note that both *SimpleDisQ* and *DisQ* achieve lower error than *NaiveAverage*. This clearly shows how combining artificial intelligence with the wisdom-of-crowds leads to improved results. Finally, It is easy to see that the average results from our algorithm are superior to those of the other algorithms. This is especially noticeable for lower B_{obj} budget but is also true for higher B_{obj}. And again, although the we show results for the average case, this is true for most cases. One can see, for example, that in order to achieve an accuracy of less than 0.067 one needs to spend 10¢ per object in *SimpleDisQ* but only 6¢ per object in our algorithm. This statement is still true after including the extra budget for the preprocess phase, since the cost of learning a regression when $B_{\text{obj}} = 10$¢ exceed the \$30 used in our algorithm for $B_{\text{obj}} = 6$¢. In figure 2 we show more examples of the budget necessary for achieving different accuracies in the different algorithms.

Other Examples. Figures 1c and 1f show equivalent graphs for a case of two query attributes (*Bmi, Age*) and figures 1b and 1e show equivalent graphs in the recipes domain (for the query attribute *Protein*). It is easy to see that all of our observations about the first example (*Bmi*) are also true when adding to it a second attribute (*Age*). In the case of *Protein*, however, this is only partly true. Consider first figure 1e. At a first glance it looks different from figure 1d. However, a closer look shows that all our observation still hold. The only difference is that *NaiveAverage* performs much worse and this changes the proportions of the graph. We believe this is because *Protein* is much less intuitive than *Bmi*. Next, consider figure 1b. Here, in addition to the different proportions we also see a different trend. Unlike the previous cases where we saw that increasing B_{obj} always decreased the error, in the case of proteins we see increase in error for $B_{\text{obj}} > 4$¢. The reason for this lies in *CollectingAttributesCondition*. Since our stopping criteria for discovering new attributes depends on B_{obj}, for higher B_{obj} we have less budget for the dismantling which in turn result in less attributes and therefore in a larger error. The effect of a smaller attributes set A_{final} also exists for *Bmi*, but in that case its effect was smaller than the effect of the increased B_{obj}. A reasonable conclusion is that for large B_{obj} budget one should also provide a large B_{prc} budget.

The experiments described above demonstrate the trends in all of our experiments: In all settings our algorithm outperforms the competing algorithms. Increasing improvements are observed when query attributes are difficult and



(a) $\mathcal{A}(Q) = \{Bmi\}$ (b) $\mathcal{A}(Q) = \{Protein\}$ (c) $\mathcal{A}(Q) = \{Bmi,\ Age\}$

(d) $\mathcal{A}(Q) = \{Bmi\}$ (e) $\mathcal{A}(Q) = \{Protein\}$ (f) $\mathcal{A}(Q) = \{Bmi,\ Age\}$

Figure 1: Error in query estimation for varying B_{prc} (top row) and varying B_{obj} (bottom row)

the relative improvement normally grows with the budget allocated to the preprocessing increases, in particular in cases when the per-object online budget is small.

5.3 Algorithm Components

We next examine the individual algorithm components. In particular we analyze our attributes dismantling method and the processing of multiple query attributes.

5.3.1 Dismantling Attributes

We considered two dimensions here.

Finding Relevant Attributes. We first tested if the crowd can give good answers to attribute dismantling questions, and if so, then how. We created gold standard attributes sets for different data domains and query attributes, and tested the crowd coverage for these attributes (percentage of discovered attributes). We computed the performance of our dismantling process and of a naive approach that asks questions only about the attributes explicitly appearing in the query. For defining the gold standard in the pictures domain (for the query attributes *Height* and *Weight*) we used the expert-provided attributes from [27]. For the gold standard in the recipes domain (for the query attributes *Proteins* and textitCalories) we used an expert dietitian.

For all queries our algorithm yielded over 80% coverage, and we compensated for the missing attributes by other discovered attributes not mentioned by the experts. This shows that the crowd could indeed replace the experts for this task. In contrast, the coverage for the naive algorithm fell below 50%, demonstrating the necessity of our choice to dismantle additional attributes. We further validated these observations by considering two additional real-life attribute domains: house prices (using [18] as a gold standard), and laptop prices (using [9]), obtaining similar results.

The GetNextAttribute Method. We next compared our technique for choosing the next attribute to dismantle to a simpler alternative where the only attributes considered are the ones appearing explicitly in the query. We call this variant *OnlyQueryAttributes*. (We also considered variations of *OnlyQueryAttributes* and *DisQ* that chose questions at random, but since those variation are very naive and were consistently inferior to our algorithm we omitted those results). Two example experiment, for the recipes domain with and the query attribute protein are shown in figure 3b (for $B_{\text{prc}} = 30$ and varying B_{obj}) and figure 3a (from $B_{\text{obj}} = 4$¢and varying B_{prc}). First, it is easy to see that our previous observations on *DisQ* in figures 1b and 1e also hold for *OnlyQueryAttributes*. Second, *DisQ* consistently outperform *OnlyQueryAttributes* illustrating again the necessity of our approach. This intensifies as B_{prc} grows since there exist enough budget to learn many attributes so the low variety of answers to the dismantling question only about protein becomes apparent. Similar trends were observed in all settings - different query attributes, query length and domains. The only thing to note is that in some specific cases, when the answers to the dismantling questions about the query attributes were varied enough the difference between the algorithms became noticeable only for large B_{prc}.

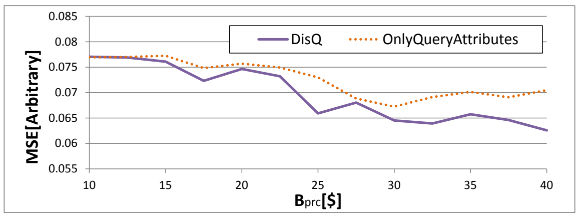
5.3.2 Statistic Estimation

We next examine our method for collecting partial statistics in queries with multiple attributes. As the main issues here are which attributes should be paired with which query attributes, and how to compensate for the missing pairs, we compared our solution to the following baselines.

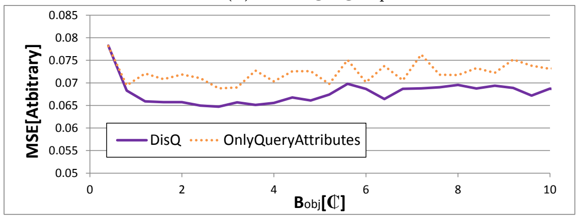
TotallySeparated - This is the naive solution that solves the problem separately for each query attribute, splitting the budget equally between them.

Full - This is a simplified variant of our algorithm that does not optimize the computation and simply gathers statistics for all attribute pairs.

OneConnection - This is another simplified variant that does consider only some of the pairs, but uses a more naive heuristic for choosing them: When a new attribute is discovered, it is paired only with one query attribute.

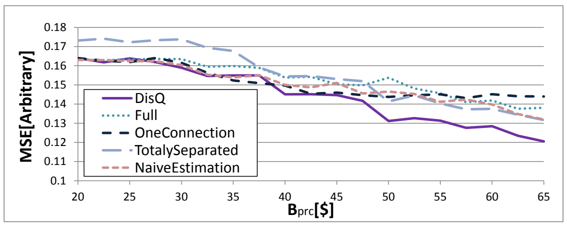


(a) Changing B_{prc}

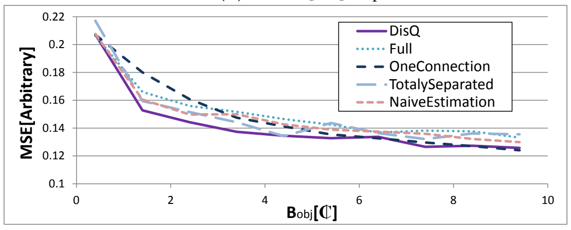


(b) Changing B_{obj}

Figure 3: Error in estimation for $\mathcal{A}(Q) = \{Protein\}$



(a) Changing B_{prc}



(b) Changing B_{obj}

Figure 4: Error in estimation for $\mathcal{A}(Q) = \{Bmi,Age\}$

NaiveEstimations - Finally, this variant selects the pairs using our technique, but rather than inferring individual values for the missing pairs, it assigns to all a default value that equal to the average S_o value.

A sample of the results, for the pictures domain and the query attributes *Bmi* and *Age* are shown in figures 4a ($B_{\text{obj}} =$ 4¢, varying B_{prc}) and figure 4b ($B_{\text{prc}} = \$50$, varying B_{obj}). We set here B_{prc} to \$50 to highlight the trends, as we will shortly discuss. First, note that all the variations follow the general trends of our algorithm as discussed above, in regard for their dependencies in B_{prc} and B_{obj}. Second, it is easy to see the relatively bad performances of the *TotallySeperated* baseline (especially for lower B_{prc}) which demonstrates the advantage of asking about several query attributes together. Third, in comparison to *Full*, our algorithm consistently achieves better (or at least as good) results when considering reasonable B_{prc} (as in figure 4a). This effect was even more noticeable in the synthetic domain where we could test large queries. For some queries, however, these trends change for very high B_{prc} as the saved budget from the non-important pairs is wasted on even more non-important new attributes. Next, in compare to *OneConnection*, our algorithm achieves at least as good results for all budgets, and better results for high B_{prc}. The reason for that is that for large budgets *OneConnection* saves budget in the beginning on redundant connections, but that budget is then referred to even more redundant attributes. In some cases, however, and for low B_{prc}, *OneConnection* did get better results, but only very marginally. The reason is the tradeoff between B_{obj} and flexible-B_{prc} we discussed before which effects *DisQ* more. Finally, our algorithm consistently outperforms *NaiveEstimations*. This is true for every budget since our estimation method incures no crowd cost.

5.4 Dependency on Assumptions

Our last set of experiments examined the robustness of our algorithm to some changes in underlying our assumptions. We briefly discuss the assumptions considered and our conclusions. A detailed description of the experiments and results can be found in our full paper [22].
Attributes Quality: We tested resilience to receiving also some irrelevant attributes in the dismantling process. This did not affect the previous trends, but as expected required somewhat higher preprocessing budget (B_{prc}) for obtaining same error rates.
Normalization Mechanism: We tested the necessity of identifying different answers about the same property as one. Here again, our algorithm can work with imperfect (or even no) unification and the trends stay the same. Somewhat higher B_{prc} is again needed for obtaining same error rates.
Answer's Correlation Parameter: We used different constants (instead of just 0.5) for $\mathbb{E}[\rho(a_j, ans_j)]$ when estimating $S_{A_{m|a_j}}$. The results remained similar.
Crowd-Tasks Payment: We tested how a different pricing models for crowd tasks impact the results. Change in prices changed some of the gradients in the varying-B_{prc} graphs but the trends remained the same.

6. RELATED WORK

Using the crowd as a source of knowledge, and for solving problems, has attracted much research in recent years [11]. The crowd was shown to be a useful tool for many types of tasks, including, but not restricted to, value estimation [24], data filtering [25], information collection [5], natural languages processing [6] etc. However, to our knowledge, our work is the first to consider using the crowd for discovering query-related attribute names. There has also been much work dealing with the collection of data via various platforms (e.g., payment [2] or games [10]), and the effective collection of such data (e.g., how to best present questions [13], how to filter spam [19], when to stop asking [30] etc.). Our work exploits these platforms and previous results. The concept of removing experts from crowd processes was also researched before[14], but not in the context of query estimation.

Our work is also influenced by previous work in machine learning, and on the use of supervised learning for regression learning (e.g., [12]). A more specific problem related to our challenge is feature selection [17] - how to effectively narrow a set of attributes for some learning process. Two models that are particularly interesting are budget learning ([23]), where the issue is deciding which is the most valuable feature to measure next under a limited budget, and meta-features (e.g., [28]), where the issue is trying to predict unseen features behavior based on some properties and similarity to other features. All of those problems, however, focus on a given predefined set of attributes. They also do not consider the selection of the same attribute name more than once (required here due the uncertainty of crowd answers).

Previous work has also dealt with the combination of crowd and learning (e.g., [8]). The common combinations are to use the crowd to label a data set (e.g., [32]) or for filling attributes values (e.g., [21], [26]).Closest to our work is that of [27] which also deals with estimating one attribute value by asking about others, but, as explained in the Introduction, requires experts-in-the-loop. While we use some of their results as basic building blocks, a major contribution here is our crowd-based attribute dismantling along with the careful statistical analysis that allows for an effective experts-free algorithm.

7. CONCLUSION AND FUTURE WORK

We studied in this paper the problem of query evaluation when the value of the queried attributes is not available in the database and is also hard for the crowd to estimate. We proposed a novel approach that uses the crowd to dismantle the query attributes into finer related ones (whose value estimation is easier), then assembles them to yield better estimation for the query attributes. Given an online per-object budget and an offline preprocessing budget, we presented an algorithm that ideally uses the offline budget for dismantling the query attributes and deriving linear formulas that best exploit the online budget for deriving the values of query attributes. We have also demonstrated the effectiveness of the approach through experimental study on both real-life crowd and synthetic data.

We focus in the paper on the minimization of the expected-mean-square-error. Other error measures may also be of interest for future research. For example, a recall-precision measurement may fit more for boolean query attributes like *gluten_free* (for recipes), or for a categorical attribute like *cousin_type* where the number of possibilities may be large. We also considered only linear formulas for assembling attributes values. While this has proved to provide good experimental results, more general rules may be useful in certain situations and we intend to study this in future work. In our development we assumed that we are given an on-line per-object budget and an offline preprocessing budget and used the later to optimize the usage of the former. Determining automatically what these budgets should be and the ideal ratio between them is an intriguing future research.

Acknowledgements

This work has been partially funded by the European Research Council under the FP7, ERC grant MoDaS, agreement 291071, and by the Israel Ministry of Science.

8. REFERENCES

[1] AllRecipes.com. `http://allrecipes.com`.

[2] Amazon Mechanical Turk. `https://www.mturk.com`.

[3] Crowd Flower. `http://www.crowdflower.com`.

[4] The height and weight chart. `http://www.cockeyed.com/photos/bodies/heightweight.html`.

[5] Y. Amsterdamer, Y. Grossman, T. Milo, and P. Senellart. Crowd mining. In *SIGMOD Conference*, 2013.

[6] C. Biemann. Creating a system for lexical substitutions from scratch using crowdsourcing. *Language resources and evaluation*, 47(1), 2013.

[7] W. Bousfield and C. Sedgewick. An analysis of sequences of restricted associative responses. *The Journal of General Psychology*, 30(2), 1944.

[8] A. Brew, D. Greene, and P. Cunningham. The interaction between supervised learning and crowdsourcing. In *NIPS workshop on computational social science and the wisdom of crowds*, 2010.

[9] P. D. Chwelos, E. R. Berndt, and I. M. Cockburn. Faster, smaller, cheaper: an hedonic price analysis of pdas. *Applied Economics*, 40(22), 2008.

[10] D. Deutch, O. Greenshpan, B. Kostenko, and T. Milo. Declarative platform for data sourcing games. In *WWW*, 2012.

[11] A. Doan, R. Ramakrishnan, and A. Y. Halevy. Crowdsourcing systems on the world-wide web. *Communications of the ACM*, 54(4), 2011.

[12] N. R. Draper and H. Smith. *Applied regression analysis 2nd ed.* New York New York John Wiley and Sons 1981., 1981.

[13] C. Eickhoff and A. P. de Vries. Increasing cheat robustness of crowdsourcing tasks. *Inf. Retr.*, 16(2), 2013.

[14] C. Gokhale, S. Das, A. Doan, J. F. Naughton, N. Rampalli, J. Shavlik, and X. Zhu. Corleone: Hands-off crowdsourcing for entity matching. In *SIGMOD Conference*, 2014.

[15] G. H. Golub and C. Reinsch. Singular value decomposition and least squares solutions. *Numerische Mathematik*, 14(5), 1970.

[16] S. B. Green. How many subjects does it take to do a regression analysis. *Multivariate behavioral research*, 26(3), 1991.

[17] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3, 2003.

[18] D. Harrison Jr and D. L. Rubinfeld. Hedonic housing prices and the demand for clean air. *Journal of environmental economics and management*, 5(1), 1978.

[19] P. G. Ipeirotis, F. Provost, and J. Wang. Quality management on amazon mechanical turk. In *Proceedings of the ACM SIGKDD Workshop on Human Computation*, 2010.

[20] L. P. Kaelbling. *Learning in embedded systems.* MIT press, 1993.

[21] E. Kamar, S. Hacker, and E. Horvitz. Combining human and machine intelligence in large-scale crowdsourcing. In *AAMAS*, 2012.

[22] M. Laadan. Dismantling complicated query attributes with crowd. Master's thesis, Tel-Aviv University, 2014.

[23] D. J. Lizotte, O. Madani, and R. Greiner. Budgeted learning of naive-bayes classifiers. *CoRR*, abs/1212.2472, 2012.

[24] J. Noronha, E. Hysen, H. Zhang, and K. Z. Gajos. Platemate: crowdsourcing nutritional analysis from food photographs. In *UIST*, 2011.

[25] A. G. Parameswaran, H. Garcia-Molina, H. Park, N. Polyzotis, A. Ramesh, and J. Widom. CrowdScreen: algorithms for filtering data with humans. In *SIGMOD Conference*, 2012.

[26] G. Patterson and J. Hays. SUN attribute database: Discovering, annotating, and recognizing scene attributes. In *CVPR*, 2012.

[27] S. Sabato and A. Kalai. Feature multi-selection among subjective features. In *ICML (3)*, 2013.

[28] B. Taskar, M. F. Wong, and D. Koller. Learning on the test data: Leveraging unseen features. In *ICML*, 2003.

[29] A. Towsley, J. Pakianathan, and D. H. Douglass. Correlation angles and inner products: Application to a problem from physics. *ISRN Applied Mathematics*, 2011.

[30] B. Trushkowsky, T. Kraska, M. J. Franklin, and P. Sarkar. Crowdsourced enumeration queries. In *ICDE*, 2013.

[31] A. Wald. Sequential tests of statistical hypotheses. *The Annals of Mathematical Statistics*, 16(2), 1945.

[32] P. Ye and D. Doermann. Combining preference and absolute judgements in a crowd-sourced setting. In *ICML (3)*, 2013.

Group Recommendation with Temporal Affinities

Sihem Amer-Yahia[†1], Behrooz Omidvar-Tehrani[†‡1], Senjuti Basu Roy[‡2], Nafiseh Shabib[‡†3]

[†] *CNRS and LIG;* [†‡]*LIG;* [‡] *University of Washington Tacoma;* [‡†] *Norwegian University of Science and Technology*

[1]{sihem.ameryahia,behrooz.omidvar-tehrani}@imag.fr, [2]senjutib@uw.edu, [3]shabib@idi.ntnu.no

ABSTRACT

We examine the problem of recommending items to ad-hoc user groups. Group recommendation in collaborative rating datasets has received increased attention recently and has raised novel challenges. Different consensus functions that aggregate the ratings of group members with varying semantics ranging from least misery to pairwise disagreement, have been studied. In this paper, we explore a new dimension when computing group recommendations, that is, *affinity between group members* and *its evolution over time*. We extend existing group recommendation semantics to include temporal affinity in recommendations and design `GRECA`, an efficient algorithm that produces temporal affinity-aware recommendations for ad-hoc groups. We run extensive experiments that show substantial improvements in group recommendation quality when accounting for affinity while maintaining very good performance.

1. INTRODUCTION

Group recommendation refers to finding the best items that a set of users will appreciate together. It is an active research area as exemplified by numerous publications [3, 6, 13, 20, 23]. The main focus of existing work in group recommendation is the design of appropriate consensus functions that aggregate individual group members' preferences to reflect the group's preference for each item. A variety of functions have been used ranging from majority voting to least misery. In this paper, we are interested in exploring *how affinity between group members* and *its evolution over time* affect group recommendations. To the best of our knowledge, our work is the first to study *affinity and its evolution over time* in combination with existing group consensus functions.

The premise of this work relies on a simple conjecture that is, a user appreciates recommendations differently in the company of different people and at different times. When with girlfriends, a female user may want to watch a romantic movie that she may not want to watch with men. When with her parents, she may prefer to go to a nice Italian restaurant while she would prefer a burger joint with her kids. In addition, her appreciation of an item with the same group of people may change over time depending on how their connection and shared interests evolve. In other terms, *the affinity of a user with other group members* should be captured in how that user appreciates an item.

Previous studies on single user recommendation have shown that *contextual dimensions* such as a user's mood and company or time and place, may affect her preferences [1, 2]. Indeed, according to behavioral research studies [5, 14, 18], consumers use different decision-making strategies and favor different brands and products depending on their context. Such observations can be incorporated in different ways into single user recommendations. In [1], a multidimensional recommendation model is developed to account for contextual information into a user's recommendation, one of which could be her affinity with other users. However, to the best of our knowledge, multidimensional recommendation has not been applied to group recommendation. In group recommendation, we conjecture that each user will have a *relative preference for an item* depending on her affinity with other group members. Formalizing the semantics of relative preference raises two new challenges: *(i) how to account for user affinities in the definition of relative preference* and *(ii) how to combine relative preference with popular group recommendation consensus functions* [13].

A major difficulty when addressing *(i)* is to integrate the evolution of affinities between users over time. For example, interns at a research lab may subscribe to a Facebook group during their internship. When the internship period is over, the group becomes an alumni of the research lab and affinities between its members will likely change. Therefore, if events such as workshops or conferences are to be recommended to the alumni group in the future, affinities between its members should be accounted for, in order to decide which subgroup would be interested in which event. While numerous recent studies have shown the importance of accounting for time in recommender systems [8, 15, 17, 25], they have focused on user-item preferences and single-user recommendations. In this work, we propose two dynamic models to capture temporal affinities: a *discrete* model where time is discretized over a set of time periods and affinities computed for each sub-period, and a *continuous* model where time is represented as an exponential function that positively or negatively affects affinity over time. Both models have a static component that denotes how close two users are in a time-independent fashion and a dynamic component that captures the drift that the affinity of a user-pair exhibits compared to the overall user population. Finally, while the discrete model is an approximation of the continuous one, they are both used to capture increasing and decreasing affinities.

Clearly, combining user-item preferences of group members independently of each other to produce group recommendations is not enough to capture the impact of affinities on those recommendations. In other terms, applying the well-known group consensus functions such as aggregated voting, average preferences or least

10.5441/002/edbt.2015.37

misery on individual group members' preferences, does not capture a scenario where the same user appreciates the same item differently in different groups. Therefore, we propose a two-step approach to address *(ii).* First, we modify individual user-item preferences on-the-fly to account for affinities and then we apply a group consensus function over the modified preferences. This approach has the benefit of dissociating recommendation computation from affinity computation and therefore being able to use relative preferences with any group recommendation consensus semantics.

No recommendation work would be complete without considering both recommendation effectiveness and efficiency. Those two dimensions raise new challenges when dealing with relative preference, namely, *(i) how to assess the quality of group recommendations?* and *(ii) how to efficiently compute affinity-aware recommendations on-the-fly for ad-hoc groups?* To address *(i)*, we build a Facebook application and generate movie recommendations using MovieLens dataset[1]. We leverage friendship and common page-likes to compute affinities and run an extensive set of experiments varying *group size*, *cohesiveness* (rating similarity between group members) and *affinity* between group members. To address *(ii)*, we develop GRECA, an algorithm that non-trivially adapts the family of threshold algorithms [10], to account for affinities between user pairs that evolve over time. GRECA leverages index structures that are extremely efficient with updates for maintaining time-variant affinities, and are used to efficiently produce the top-k recommended itemset for a group. In fact, as affinity between users evolves over time, GRECA does not need to recalculate any of the previously calculated affinities and just augments the index to account for the latest affinities. In addition to being instance optimal, the key novelty of GRECA is the use of a new buffer condition for termination, which constitutes a clear departure from traditional top-k style algorithms [10]. This condition simply implies that just by examining the items in the buffer, GRECA can terminate with the guarantee to have found the correct top-k itemset.

Our experiments consistently indicate that incorporating temporal affinities into group consensus functions is most effective for dissimilar user groups as well as low-affinity user groups whose preference significantly evolves over time. Prior work has shown that such groups generally take longer to reach consensus [20]. Our performance experiments demonstrate that GRECA achieves a save up of 75% or beyond in the number of accesses. These results strongly corroborate the effectiveness of our proposed solutions to include temporal affinities in group recommendation functions.

The paper makes the following technical contributions:

- We motivate the need to account for user affinities between group members when computing recommendations and propose to capture affinities in *the relative preference* of individual group members for each item. Relative preference modifies a user-item preference with the user's affinity with other group members.

- Since affinities may evolve over time, we propose two models, discrete and continuous, to represent (positive or negative) affinity drift of two users over time. This dynamic component is combined with a static component, that captures how close two users are in a time-independent fashion, in order to form temporal affinities.

- We extend group recommendation semantics, i.e., average preferences, least misery and pairwise disagreement, to include temporal affinities and design GRECA, an efficient algorithm that computes recommendations on-the-fly for ad-hoc groups. GRECA uses a new early termination condition to efficiently produce the top-k itemset for a group.

- We run extensive experiments using Facebook and MovieLens datasets and examine the impact of our temporal affinity model on group recommendation quality and efficiency.

The paper is organized as follows. Section 2 contains our formalism. GRECA, our recommendation algorithm is provided in Section 3. Extensive experimental evaluation is given in detail in Section 4. Related work is summarized in Section 5 and conclusion in Section 6.

2. DATA MODEL AND PROBLEM

We present a data model that captures temporal affinities and define our problem of recommending items to ad-hoc groups.

The underlying scenario that will be used to illustrate our model is a social network of individuals who have some intrinsic characteristics (e.g., birthplace, gender and age) and who express interests for items via likes and votes as in Facebook and Twitter. At any given point in time, we are interested in recommending content items (e.g., movies, books, conferences) to an ad-hoc group. Parts of this scenario will be used in this section and one instance will be described in specific details in Section 3.1.

In our model, we assume a set of m items $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ and a set of n users $\mathcal{U} = \{u_1, \ldots, u_n\}$ out of which any ad-hoc group $\mathcal{G} \subseteq \mathcal{U}$ can be built. To simplify exposition, we will not formalize user or item attributes and will refer to them when needed in our example. We consider time as a set of consecutive timestamps that form periods. Each period p is a time interval of the form $[s, f]$ where s is its starting timestamp and f its ending timestamp.

2.1 Dynamic User Affinity Models

Affinity describes the *bonding* between a pair of users (u, u') and is denoted $aff(u, u')$. It could be as simple as explicit friendship or users in the same age group or more sophisticated such as users who like similar movies, have visited similar places and have friends who live in different parts of the world. For simplicity, we assume that affinity between a user pair is symmetric, i.e., $aff(u, u') = aff(u', u)$. More importantly, $aff(u, u')$ is dynamic and changes over time. We therefore compute affinity $aff(u, u', p)$ for a time period $p = [s, f]$. This dynamic affinity captures changes over time by combining its *static* and *dynamic* components defined below.

- *Static Affinity* - $aff_S(u, u')$: This is a time-agnostic affinity component and is used to capture how close two users are in a time-independent fashion. Stable factors such as birthplace, age, and education naturally contribute to this component. However, depending on the application, other dimensions could be accounted for. For example, Facebook friendship being stable, we use it to model static affinity in our experiments (Section 4.1.2).

- *Dynamic Affinity* - $aff_V(u, u', p)$: This is a time-variant component that captures affinity between two users u and u' during period p by considering how close they are during that period. For example, shared political interests, common likes, and shared interests for world events, vary over time and could contribute to formulating this component. Intuitively, the objective is to capture the *aggregated drift* that the affinity of a user pair exhibits for every time period from the beginning of time s_0 to the end of the current period $p = [s, f]$, compared to the overall user population.

[1] *http://movielens.umn.edu/*

More specifically, time starts at the beginning of time s_0 and is segmented into subsequent time periods $p_0, \ldots, p_{now}$ of varying lengths. Given two time periods $p_i = [s_i, f_i]$ and $p_j = [s_j, f_j]$, $p_i \leq p_j$ is used to denote that p_i precedes p_j, i.e., $s_i \leq s_j$ and $f_i \leq f_j$. Determining the right granularity of a time period depends on the application at hand and the frequency of user actions and is orthogonal to our model. For example, in a social network such as Facebook, and when affinities are computed using shared posts, granularity may vary from hours to days depending on the time of year. On Twitter, granularity is finer and may vary from minutes to hours since post frequency is higher. Not all time periods are of the same length.

Given a time period $p = [s, f]$, for every time period p' that is included in the interval starting at the beginning of time s_0 and ending at f, the end of p, the periodic affinity drift is calculated as a difference between the periodic affinity $\mathit{aff}^P(u, u', p')$ between users u and u' and the average periodic affinity $\mathit{Avgaff}^P(p')$ of the whole user population. These drifts are aggregated over all time periods included in the interval $[s_0, f]$ and normalized to generate $\mathit{aff}_V(u, u', p)$. Formally,

$$\mathit{aff}_V(u, u', p) = \frac{\Sigma_{p' \leq p}(\mathit{aff}^P(u, u', p') - \mathit{Avgaff}^P(p'))}{\Delta} \quad (1)$$

The exact formulation of Δ depends on how time is modeled (discrete or continuous) and is described. Interestingly, $\mathit{aff}_V(u, u', p)$ could be either positive or negative and depends on how the affinity of (u, u') evolves compared to the overall population.

The exact formulation of $\mathit{aff}^P(u, u', p')$ depends on the application. In our Facebook experiment in Section 4.1.2, we use common page likes between u and u' during period p'.

Finally, $\mathit{Avgaff}^P(p')$ is defined as follows:

$$\mathit{Avgaff}^P(p') = \frac{2 \times \Sigma_{(u,u') \in \mathcal{U}, u \neq u'} \mathit{aff}^P(u, u', p')}{|\mathcal{U}|^2 - |\mathcal{U}|}$$

We now describe our dynamic affinity models that use $\mathit{aff}_S(u, u')$ and $\mathit{aff}_V(u, u')$ as building blocks. The first model relies on discretized time periods to capture aff_V whereas the second represents time in a continuous fashion.

- *Discrete Dynamic Affinity Model:* In this model, the aff_S and aff_V affinity components are aggregated using a linear function over a set of discretized time periods. Therefore, Δ, the denominator in Equation 1, is simply the number of time periods between s_0, the beginning of time, and e, the end of p. This simple aggregation also allows us to design efficient algorithms.

  $\mathit{aff}^D(u, u', p) = \mathit{aff}_S(u, u') + \mathit{aff}_V(u, u', p)$

- *Continuous Dynamic Affinity Model:* For this model, time is considered in a continuous fashion. In this case, the denominator in Equation 1, Δ=f-s_0 is the length of time between the the beginning of time s_0 and f, the end of p. As a natural representation to capture continuous time, we consider an exponential function, which is also supported in prior work [17]. Formally,

  $\mathit{aff}^C(u, u', p) = \mathit{aff}_S(u, u') \times e^{\lambda(f - s_0)}$

  Here λ is the rate of growth/decay of affinity and could simply be replaced by $\mathit{aff}_V(u, u', p)$ in Equation 1 to represent the cumulative effect of affinity drift over time.

Consequently, the discrete time model could be viewed as an approximation of the continuous one where time is discretized into sub-periods and each user pair's affinity is normalized over the number of periods (Equation 1). Alternatively, the continuous model treats time as a single interval $[f$-$s_0]$ and captures an exponential growth, resp., decay, affinity model when $\mathit{aff}_V(u, u', p)$ is positive, resp., negative.

In both $\mathit{aff}^D(u, u', p)$ and $\mathit{aff}^C(u, u', p)$ affinity drift could be negative or positive thereby capturing situations in practice where affinity between two users may increase or decrease over time. We believe that the ability to capture this varying rate of change is important in practice in particular for social networks where different users exhibit different interests over time.

2.2 User-Item Preference Models

We now show how affinities are accounted for in computing the preference of a user for an item in a group. We first describe how affinity is incorporated into user-item preferences without accounting for time then we show how to modify the formulation to compute time-aware user-item preferences.

Time-Agnostic User-Item Preference: Given a group $\mathcal{G}$, the preference of a user $u \in \mathcal{G}$ for an item $i \in \mathcal{I}$ is denoted $\mathit{pref}(u, i, \mathcal{G})$ and depends on two components:

- *Absolute preference - $\mathit{apref}(u, i)$.* This describes how much u likes item i akin to the predicted rating of u for i. Existing single-user recommendation algorithms, such as collaborative filtering, could be used to compute $\mathit{apref}(u, i)$.

- *Relative preference - $\mathit{rpref}(u, i, \mathcal{G})$.* This component captures that *a user likes an item i if close members in the group $\mathcal{G}$ also like i* and similarly that *a user dislikes an item i if close members in the group $\mathcal{G}$ dislike i*. Affinity between group members is used to capture how close they are. More formally, $\mathit{rpref}(u, i, \mathcal{G})$ combines the affinity of a user u with other members $u' \in \mathcal{G}$, denoted $\mathit{aff}(u, u')$, with the preference of u' for item i, denoted $\mathit{apref}(u', i)$.

  $$\mathit{rpref}(u, i, \mathcal{G}) = \Sigma_{\forall u' \neq u \in \mathcal{G}} \mathit{aff}(u, u') \times \mathit{apref}(u', i)$$

The overall affinity-aware user-item preference is a simple combination of these two factors: $\mathit{pref}(u, i) = \mathit{apref}(u, i) + \mathit{rpref}(u, i, \mathcal{G})$

Time-Aware User-Item Preference: We now modify the definition of relative preference to capture temporal affinities:

$$\mathit{rpref}(u, i, \mathcal{G}, p) = \Sigma_{\forall u' \neq u \in \mathcal{G}} \mathit{aff}(u, u', p) \times \mathit{apref}(u', i).$$

Therefore, a user u's overall preference on item i during time period p can be simply formulated as:

$$\mathit{pref}(u, i, \mathcal{G}, p) = \mathit{apref}(u, i) + \mathit{rpref}(u, i, \mathcal{G}, p)$$

2.3 Group Consensus Models

Members of a group may not always have the same preferences for items and a *consensus function* needs to aggregate user-item preferences into a single group's preference for an item. Intuitively, there are two main aspects in a consensus function [20]. First, the preference of a group for an item needs to *reflect the degree to which the item is preferred by all group members*. The more group members prefer an item, the higher its group preference. Second, the group preference needs to *capture the level at which members disagree or agree with each other.* All other conditions being equal, an item that draws high agreement should have a higher score than an item with a lower overall group agreement. We call the first aspect *group preference* and the second aspect *group disagreement.* We revisit the definitions we introduced in [3] to include a time component.

Group Preference: The preference of an item i by a group $\mathcal{G}$ during a time period p, denoted $gpref(\mathcal{G}, i, p)$, is an aggregation over the preferences of each group member for that item. We consider two commonly used aggregation strategies:
Average Preference: $\frac{1}{|\mathcal{G}|}\sum_{u\in\mathcal{G}}(pref(u,i,\mathcal{G},p))$
Least-Misery Preference: $min_{u\in\mathcal{G}}(pref(u,i,\mathcal{G},p))$

Group Disagreement: The disagreement of a group $\mathcal{G}$ over an item i during a time period p, denoted $dis(\mathcal{G}, i, p)$, reflects the degree of consensus in the user-item preferences for i among group members over time. We revisit the two most common disagreement computation methods:

1) *Average Pair-wise Disagreements:*
$dis(\mathcal{G},i,p) = \frac{2}{|\mathcal{G}|(|\mathcal{G}|-1)}\sum(|pref(u,i,\mathcal{G},p) - pref(v,i,\mathcal{G},p)|)$, where $u \neq v$ and $u, v \in \mathcal{G}$;

2) *Disagreement Variance:*
$dis(\mathcal{G},i,p) = \frac{1}{|\mathcal{G}|}\sum_{u\in\mathcal{G}}(pref(u,i,\mathcal{G},p) - mean(i,\mathcal{G},p))^2$, where $mean(\mathcal{G},i,p)$ is the mean of all the individual preferences for item i over time.

The average pair-wise disagreement function computes the average of pair-wise differences in preferences for the item among group members, while the variance disagreement function computes the mathematical variance of the preferences for the item among group members. Intuitively, the closer the preferences for i between users u and v, the lower their disagreement for i.

Time-Aware Group Consensus: We combine group preference and group disagreement in a time-aware consensus function, denoted $\mathcal{F}(\mathcal{G}, i, p)$. The function combines group preference and disagreement for an item i and a group $\mathcal{G}$ into a single group consensus score using the following formula: $\mathcal{F}(\mathcal{G},i,p) = w_1 \times gpref(\mathcal{G},i,p) + w_2 \times (1 - dis(\mathcal{G},i,p))$ where $w_1 + w_2 = 1.0$ and each specifies the relative importance of preference and disagreement in the overall group consensus.

Note that the formulation of group consensus incorporates temporal affinities by aggregating the relative user-item preferences of its members with disagreement. The proposed formulation is orthogonal to how affinities are modeled and incorporated into individual relative preferences. This way accounting for temporal affinities in group recommendation is orthogonal to the consensus function used to aggregate group members.

2.4 Problem Definition

Given a group $\mathcal{G}$, a time-aware consensus function $\mathcal{F}$ and an integer k, the objective is to recommend to $\mathcal{G}$ the k best itemset $\mathcal{I}_G$ that accounts for its members' affinities during a period p, such that:

- $|\mathcal{I}_\mathcal{G}| = k$
- $\forall i \in \mathcal{I}_\mathcal{G}, u \in \mathcal{G}$, i is not individually recommended to u
- $\nexists j \in \mathcal{I}$, s.t. $\mathcal{F}(\mathcal{G},j,p) > \mathcal{F}(\mathcal{G},i,p)$, where $j \notin \mathcal{I}_\mathcal{G}, i \in \mathcal{I}_\mathcal{G}$, i.e., there does not exist any other item j in $\mathcal{I}$ whose consensus score is higher than any item in i in $\mathcal{I}_\mathcal{G}$.

3. INSTANCE OPTIMAL ALGORITHMS

In this section, we discuss how to efficiently compute k affinity-aware recommendations for ad-hoc groups, meaning for groups that are not known beforehand. Recall that given a group $\mathcal{G}$, the goal, stated in Section 2, is to find the k best items to recommend to $\mathcal{G}$ according to a consensus function $\mathcal{F}$.

We propose instance optimal algorithms to compute top-k items for a given group under different group consensus functions. The overall intuition of this algorithm is appropriately adapted from the family of Fagin-style top-k algorithms [10]. These algorithms, such as, Threshold algorithms `TA` or No Random Access Algorithm `NRA`, rely on a function that aggregates multiple score components into a single score for each item. Those algorithms are used in Web search to compute the score of each item (a document in that case) as a combination of its component scores (its scores for each keyword in the search query). These algorithms aim to find the k items that rank the highest (the ones with the highest aggregated scores) in as little time as possible. They take sorted item lists that correspond to each component and scan them using *sequential* and *random accesses* (`SA`s and `RA`s), and the computation can be terminated without scanning the input lists fully, using stopping conditions based on score bounds (thresholds). Early stopping is possible when the ranking function is *monotone* [10].

LEMMA 1. *The temporal affinity-aware consensus function $\mathcal{F}$ is monotonic w.r.t. absolute preference lists and user-affinity lists for the dynamic user-affinity model, and pair-wise disagreement lists.*

PROOF. (sketch): In a prior work [3], we showed that all three group consensus functions without considering time-agnostic affinity (average preference, least misery and pair-wise disagreement) are monotone. If all group members, except a user u, rate items i_1 and i_2 the same, i_1 will have at least the same group preference as i_2 if u rates i_1 no less than i_2. This holds for both the average and least-misery. For pair-wise disagreement, we showed that our group disagreement functions (pair-wise and variance) can be transformed into aggregations of individual pair-wise disagreements and become monotone.

Monotonicity remains true with the introduction of affinities and time. For an item i, if both users like i highly, higher affinity between them only improves i's overall preference. On the contrary, for an item j, if they like j as highly as they do i, lower affinity between them only decreases j's overall preference. Introduction of time in the affinity model only makes the affinity calculation time-dependent by changing the temporal granularity at which it is computed; however, the relationship between dynamic affinity and the group consensus of an item does not change. □

As a result, we can design instance optimal algorithms with the early stopping.

3.1 Running Example and Data Structures

We now describe the data structures necessary to run Fagin-style top-k processing algorithms via an example that will also be used to illustrate our algorithm, `GRECA`.

Imagine a group $\mathcal{G}$ formed with three users u_1, u_2, u_3. Given an itemset $\mathcal{I} = \{i_1, i_2, i_3\}$, our objective is to identify the best item ($k = 1$) to recommend to the group at time period p (for example, January 2014). Also, assume that the system has information about

Algorithm 1 Group Recommendation with Temporal Affinities (GRECA)

```
Require: Group $\mathcal{G}$, $k$, consensus function $\mathcal{F}$;
1: Retrieve user preference lists $\mathcal{PL}_u$ for each user $u$ in group $\mathcal{G}$;
2: Retrieve pair-wise affinities for $\mathit{aff}_S$, $\mathit{aff}_V$ for each period;
3: $Sc_r = \{r_u\}$, the last user preference from $\mathcal{PL}_u, \forall u \in \mathcal{G}$
4: $Sc_{\mathit{aff}_S} = \{\mathit{aff}_{S\,u,v}\}$, the last pair-wise $\mathit{aff}_S$ affinity values read $\forall u, v \in \mathcal{G}$
5: $Sc_{\mathit{aff}_V} = \{\mathit{aff}_{V\,u,v}\}$, the last pair-wise periodic affinity values read for each time period $p'$, $\forall u, v \in \mathcal{G}$
6: Cursor $cur$ = getNext() round-robin accesses to $\mathcal{PL}_u$, $\mathit{aff}_S$ and $\mathit{aff}_V$ lists
7: while ($cur <>$ NULL) do
8:   Get entry $e$ at $cur$
9:   if !(in $\mathcal{B}$(topKHeap, $e$)) then
10:    ComputeUB(e, $\mathcal{F}$)
11:    ComputeLB(e, $\mathcal{F}$)
12:    Add $e$ in $\mathcal{B}$
13:  else
14:    Update ComputeUB(e, $\mathcal{F}$) and ComputeLB(e, $\mathcal{F}$)
15:  end if
16:  $Sc_{th}$ = ComputeTh({E}, $\mathcal{F}$) considering all current cursor positions
17:  if $Sc_{th} \leq \mathcal{B}.k_{th}LB \& |\mathcal{B}| = k$ then
18:    return topKList($\mathcal{B}$, ||);
19:    Exit;
20:  else
21:    if CheckBuffer ($\mathcal{B}$) is satisfied then
22:      return topKList($\mathcal{B}$, ||)
23:      Exit;
24:    else
25:      $cur$ = getNext()
26:    end if
27:  end if
28: end while
29: return topKList($\mathcal{B}$, ||)
```

group members u_1, u_2, u_3 for one year, i.e., January 2013 to January 2014. The user-item preference lists of those group members are provided in Table 1. Each list contains items preferred by each user sorted in decreasing order of preference.

The item preference of a member u of a group G is a combination $\mathit{pref}(u, i, \mathcal{G}, p) = \mathit{apref}(u, i) + \mathit{rpref}(u, i, \mathcal{G}, p)$, where *rpref* is the relative preference that accounts for the temporal affinity of u with other group members.

u_1		u_2		u_3	
i_1	5	i_1	5	i_3	2
i_2	1	i_2	1	i_1	2
i_3	1	i_3	0.5	i_2	1

Table 1: Absolute Preference Lists $\mathcal{PL}_u$ of u_1, u_2, u_3

Affinity between users consists of two components, static affinity aff_S and dynamic affinity aff_V. The detailed interpretations of these affinities and how they are calculated are given in Section 2. Out of the two aforementioned affinities, the latter is time-aware, where time is considered in a continuous fashion or over a discrete set of time periods (for example, two equal periods p_1 and p_2 each of six months in our case). For simplicity, we consider the discrete model in this example. The aff_S affinity involves all user-pairs thereby creating $3 \times (3-1)/2$ (i.e. $n(n-1)/2$ in general) entries. For every time period p', similarly, there is a periodic affinity list of the same size. Notice that each affinity list aff_V or aff_S with $n(n-1)/2$ entries could further be partitioned into a set of $n-1$ lists, where the i-th list stands for user u_i with $n-i$ entries. For example, we can have a $\mathcal{L}_{\mathit{aff}_S}(u_1)$ that stores u_1's static affinity with u_2 and with u_3, one for $\mathcal{L}_{\mathit{aff}_S}(u_2)$ with u_2's affinity with u_3 only (storing u_1's affinity here again is redundant), and no static affinity list needs to be created for user u_3. This partitioning allows us to design efficient algorithms, as we describe later in Section 3. Table 2 contains aff_S affinity lists of all users sorted in decreasing order and Tables 3 and 4 contain aff_V affinity of users in periods p_1 and p_2 respectively. Note that the temporal affinity of users u_1 and u_2 has decreased between periods p_1 and p_2.

In the above example, temporal affinity between user pairs (u, u') is modeled in a discrete manner as $\mathit{aff}^D(u, u', p)$. To facilitate efficient computation, it is easy to see that the different absolute preference lists and time-variant affinity lists are to be pre-computed. Even for a small group such as the one in the example with 3 users, there are 3 absolute preference lists. Furthermore, the all-pair user affinities for a given time period p' are to be stored as well, either as a single list with $n(n-1)/2$ entries, or decomposed over a set of $n-1$ lists, where the i-th list represent user u_i's affinities with $n-i$ other users. Since the period affinities are independent of each other, we must precompute such lists for every time period. For the example case, this requires either creating 2 periodic affinity lists to capture aff_V affinity and one static affinity list to capture aff_S. Each of these lists have $n(n-1)/2$ entries (as a single list or splitted in $n-1$ lists, as described in the example). The size of each list is quadratic in the number of users, but the number of such lists ($\mathcal{T}$) is a function of how time is discretized into periods. Even for a small group such as ours, many lists are to be used in the computation. Notice that all these user-affinity lists are required to compute the *complete score* of any item, because, the relative preference $\mathit{rpref}(u, i, \mathcal{G}, p)$ for every item requires accessing *all* $\mathcal{T} \times n(n-1)/2$ entries. An algorithm such as TA must read all those entries to compute the *complete score* of an item and will hence incur a large number of RAs.

We argue that all these accesses are not always necessary. For instance, based on preferences in Tables 1 to 4, we consider scanning item i_1 in $\mathcal{PL}_{u_1}$. If we were following TA method, to compute the complete score of this item, 21 RAs are needed, i.e., one RA for each $\mathit{apref}(u, i_1)$ component and 6 RAs for each $\mathit{rpref}(u, i_1, p)$ component where $u \in \{u_1, u_2, u_3\}$. Note that to compute the score of a single item i_1, we have accessed *all* entries in $\mathit{aff}_S(u_1)$, $\mathit{aff}_V(u_1, p_1)$ and $\mathit{aff}_V(u_1, p_2)$ lists. For instance, entries in the list $\mathit{aff}_S(u_1)$ is the static affinity scores between (u_1, u_2) and (u_1, u_3) where we have accessed both.

Instead, our instance optimal algorithm GRECA makes *only sequential accesses, i.e.,* SA*s* like NRA and *potentially avoids consuming all these* $\mathcal{T} \times n(n-1)/2$ *entries* to determine the top-k itemset. Following previous example, if for instance $\mathit{aff}_S(u_1, u_3)$ (in Table 2) is not yet scanned, we avoid making an RA to get this value, but based on NRA principle, we use the score under the cursor in the list of Table 2 (i.e., initially $\mathit{aff}_S(u_1, u_2)$) to compute a partial score for i_1. Details are mentioned in Section 3.2.

GRECA returns the top-k itemset which contains the best set of k-items, although the rank among the returned itemset may not be fully distinguishable (i.e. giving rise to a partial order). This is rather reasonable, because k is usually small, and the group is potentially interested in all of the k-items.

3.2 GRECA

For ease of exposition, we describe GRECA using the simplest group consensus function *Average Preference* considering time-aware affinity. The other group consensus functions mimic its behavior. The algorithm exploits the settings as is described in the example in Section 3.1.

u_1		u_2	
u_1u_2	1	u_2u_3	0.3
u_1u_3	0.2		

Table 2: Static Affinity Lists $\mathcal{L}_{\mathit{aff}_S}$

u_1		u_2	
u_1u_2	0.8	u_2u_3	0.2
u_1u_3	0.1		

Table 3: $\mathcal{L}_{\mathit{aff}_V}$ Lists for Period p_1

u_1		u_2	
u_1u_2	0.7	u_2u_3	0.1
u_1u_3	0.1		

Table 4: $\mathcal{L}_{\mathit{aff}_V}$ Lists for Period p_2

Without loss of generality, for a given group with n users, GRECA uses n user-item preference lists, where each list $\mathcal{PL}_u$ for user u has m items that are sorted in decreasing user-item preference. Each $\mathcal{PL}$ can be obtained with any single user recommendation strategy (in our experiments in Section 4, we use collaborative filtering). In addition, GRECA uses $n-1$ static affinity lists, and another $n-1$ dynamic periodic affinity lists for each time period.

The algorithm runs in a round-robin fashion over the aforementioned lists by making only SAs. It reads an entry $e = (i, r)$, where i is the item-id and r is the user u's absolute preference score for i, or an entry $e' = (u', r')$, where r' is the pair-wise affinity of (u, u'). Affinity between a pair of users is either static or periodic (i.e. dynamic), and the computation does not distinguish between these two kinds. The algorithm invokes the following 3 different subroutines to determine whether to continue further or to safely terminate and return the top-k itemset during its execution:

(a) Compute Upper-Bound of an Item: `ComputeUB(i)`**:** UB_i computes the highest score that an item i can have in $\mathcal{G}$ based on so far accesses.

(b) Compute Lower-Bound of an Item: `ComputeLB(i)`**:** LB_i computes the lowest score that an item i can have in $\mathcal{G}$ based on so far accesses.

(c) Compute Global Threshold: `ComputeTh`$(\{E\})$**:** Input to this function is the current set $\{E\}$ of entries read from all the lists. The output is simply a numeric score that captures the highest score that an *unseen* item can have for group $\mathcal{G}$.

Subroutines can be invoked after reading one entry from each type of list (preference list, static affinity list or dynamic affinity list) to make sure all types of lists are visited before or after reading the j-th entry from all lists.

The first two subroutines return the latest bounds of an item i. Then, those updated bounds are pushed into an *item buffer* that is maintained throughout the execution of the algorithm. We describe our proposed buffer management strategy later on. Naturally, these two subroutines are to be invoked for all encountered items so far.

Illustration of the Subroutines: The upper-bound score of an item i is simply the highest score it can have on current accesses. It is computed by combining the actual encountered values for some of the entries and then assigning the current cursor readings to the rest. Consider our three-user group described in Section 3.1 and assume that `ComputeUB`(i_3) is invoked after the cursor reads the second entry at $\mathcal{PL}_{u_2}$. At that point, $\mathit{apref}(u_3, i_3) = 2$ is encountered, but for the other two users, these are to be approximated based on the current cursor readings. For example, the highest score of $\mathit{apref}(u_1, i_3) = 1$, $\mathit{apref}(u_2, i_3) = 1$. Similarly, static and dynamic periodic affinities of users u_1, u_2 and u_2, u_3 are encountered, but those of u_1, u_3 are to be guessed based on the latest cursor reading from the respective lists. This gives rise to
`ComputeUB`$(i_3) = \Sigma_{\forall i \in \{1,2,3\}} \mathtt{UB}[\mathit{apref}(u_i, i_3)] + \mathtt{UB}[\mathit{rpref}(u_i, i_3, \mathcal{G}, p)]$ = 13.02 (by ignoring normalization and final averaging).

The computation of the lower-bound of an item i is similar except that it replaces the unseen entries of the function with the lowest possible score. For example, instead of assigning $\mathit{apref}(u_1, i_3) = 1$, $\mathit{apref}(u_2, i_3) = 1$, it will consider those values to be 0 (assuming that the smallest absolute preference for an item could be 0). The same will happen in affinity calculation; as an example, it substitutes $\mathit{aff}_S(u1, u_3) = 0$, instead of 0.8 in the upper-bound computation case. When invoked using item i_1, `ComputeLB`(i_1) returns a value of 14.2 (ignoring normalization and final averaging).

Computation of `ComputeTh`$(\{E\})$ is rather simple. It simply incorporates each of the entries in $\{E\}$ in the function and returns a numeric score.

Buffer Management Strategy: Once the upper-bound and lower-bound scores of each item are computed, they are pushed into a buffer $\mathcal{B}$ and are sorted in decreasing order of lower-bound score. The buffer is implemented as a heap data structure which allows efficient updates since it requires to maintain sorted lists of potential results and, in some cases, item lower-bounds and upper-bounds need to be updated (for example, when the item is encountered again in one of the lists).

Stopping Condition: The algorithm has both global threshold computation and buffer management strategies. We now show that *the buffer management* itself is sufficient to govern early stopping. More importantly, unlike traditional threshold algorithms, GRECA cannot terminate only based on the threshold condition in the cases, where the buffer contains more than k items.

- **Using the Global Threshold:** If the current global threshold is not larger than the lower-bound score of the k-th item in the buffer, GRECA will not find any item later on whose score is larger than the current threshold. On the other hand, if the current threshold is no larger than the lower-bound of the k-th item in the buffer, any unseen item can never be in the top-k itemset. This implies that a subset of the items in the current buffer is the actual top-k itemset. If the buffer contains k items only, then GRECA can safely terminate and return those items in the buffer as the answer. However, in general, when the buffer has more than k-items, to precisely determine the actual top-k itemset, it needs to apply the buffer management strategy that we describe now.

- **Using the Buffer:** A key novelty of GRECA is in using only the buffer condition for termination. This condition simply implies that just by looking into the items in the buffer, GRECA can terminate, as well as declare the partially ordered correct top-k itemset. The buffer stopping condition works as follows: the buffer contains k'-items ($k' > k$) such that the lower-bound of the k-th item score is no smaller than the upper-bound score of each of the remaining $k' - k$ items. In that case, those remaining $k' - k$ items could be safely pruned. Interestingly, satisfying this condition implies satisfying the threshold condition as well, as Theorem 1 states. The remaining k items are returned as answers.

- **Global Threshold and Buffer Management:** Global threshold can simply determine that the current buffer contains a subset of items which are the actual top-k itemset. In a general case, where the buffer has more than k items, GRECA applies the buffer stopping conditions to determine that subset. It is still possible that the buffer condition for stopping is not met. In that case, GRECA resumes computation until the buffer condition is satisfied or all lists are exhaustively scanned.

Theorem 1. Satisfying the buffer condition for termination implies that the global threshold condition for termination is met.

PROOF. (sketch): At a given snapshot during the execution of GRECA, the score returned by ComputeTh($\{E\}$) is strictly not greater than the upper-bound score of any item that is already seen and in the buffer, i.e., ComputeUB(i) $\geq$ ComputeTh($\{E\}$). Therefore, if the buffer condition is satisfied (meaning that the lower bound of the k-th item score in the buffer is not smaller than the upper-bound score of the remaining $k' - k$ items), this automatically implies, that the lower bound of the k-th item score is not smaller than the current global threshold. Hence the proof. $\square$

For our running example in Section 3.1, this returns i_1 as the top-1 item to the group.

The pseudocode of GRECA is presented in Algorithm 1. In addition to the group $\mathcal{G}$ and k, it takes the preference and affinity lists of $\mathcal{G}$ as inputs as well as the consensus function $\mathcal{F}$. Lines $9 - 14$ either add a new item into the buffer $\mathcal{B}$ and compute its lower-bound and upper-bound scores, or update the latest lower-bound and upper-bound score of an existing item and reorganize the buffer. Line 16 computes the global threshold condition using the function ComputeTh(); lines $17 - 19$ checks if the threshold stopping condition is satisfied. Otherwise, the control goes on to line 21 on wards and CheckBuffer($\mathcal{B}$) checks whether the stopping condition is met using the buffer. The computation continues unless one of these conditions are satisfied, or all lists are exhaustively scanned. Of course, in the latter case, there is no save-up. However, as our experimental results exhibits, GRECA achieves speed-up, compared to its naive counterpart.

LEMMA 2. GRECA *returns correct top-k itemset.*

PROOF. Notice that GRECA returns from the buffer those k-items whose lower-bound scores are the highest and larger than the upper-bound score of any remaining item. As Theorem 1 proves that this also implies that the global threshold at that point cannot be larger than the lower-bound score of the k-th item in the buffer. Notice that the threshold captures the highest score that any unseen item can have. Due to the monotonicity property of the consensus function, global threshold decreases gradually, implying that the highest score of any item gets only smaller, as more entries are scanned from the lists. Therefore, when GRECA terminates and outputs the itemset with the highest top-k lower-bound scores, this implies that any other items that are discarded or unseen cannot have higher score than the returned itemset. Hence the proof. However, since the complete score of many of the items may not be computed upon termination, the output may give rise to a partial order among the top-k items. $\square$

LEMMA 3. GRECA *is instance optimal.*

PROOF. (sketch): In [10], authors prove that NRA is instance optimal with optimality ratio m and no deterministic algorithm can perform any better. GRECA mimics the cursor movement of traditional NRA, however, it has a different stopping condition. Theorems 1 and 2 prove that our stopping condition implies both the threshold stopping condition and result correctness, therefore, the instance optimality of GRECA holds. A detailed proof is deferred to an extended version of the paper. $\square$

4. EXPERIMENTS

We evaluate our group recommendation method from two major angles: effectiveness and efficiency. We conduct an extensive user study on Facebook to demonstrate that group recommendation with the consideration of temporal affinity is superior to solely relying on aggregating individual preferences (Section 4.1). We also run comprehensive experiments to show that GRECA achieves scalable performance when computing temporal affinity-aware recommendations for ad-hoc groups (Section 4.2).

We implement our prototype system using JDK 1.8.0. All scalability experiments are conducted on an 2.4 GHz Intel Core i5 with 8 GB of memory on OS X 10.9.5 operating system.

Dataset Description: We use the MovieLens 1M ratings dataset [2] for our evaluation. MovieLens is a collaborative rating database where users provide a rating ranging from 1 to 5 for movies (5 being the best). Table 5 contains the statistics of the 1M ratings dataset.

# users	# movies	# ratings
6,040	3,952	1,000,209

Table 5: The MovieLens 1M Dataset

Individual User Preferences: We use collaborative filtering [2] to generate individual user preferences where user similarity is computed with cosine similarity over $vec(u)$, i.e., the ratings of u for each movie.

$$\cos(\vec{u}, \vec{u'}) = \frac{\vec{u} \times \vec{u'}}{||\vec{u}||^2 \times ||\vec{u'}||^2}$$

4.1 Quality Experiment

We exploit the availability of Facebook users for our user study which gives us the opportunity to obtain preferences of real users and leverage the social graph for affinities. Our aim is to compare our temporal affinity-aware group recommendation with naive methods without consideration of time or affinity. Our group recommendations are produced and compared using the following consensus functions (as discussed in Section 2).

- **Average Preference (AP)**, which computes the group preference for an item as the average of individual group members' preferences for that item.

- **Least-Misery Only (MO)**, which computes the group preference for an item as the minimum among individual group members' preferences for that item.

- **Pair-wise Disagreement (PD)**, which computes the group preference for an item as the combination of its average and its pair-wise disagreement between individual group members' preferences.

[2] *http://movielens.umn.edu*

For each of these functions, we incorporate time-aware affinity to compute the relative user preference to an item at a given time (see Section 2 for an exact definition of relative preference.)

We developed an application using the Facebook API[3] and recruited 72 Facebook users overall to rate movies from the MovieLens 1M dataset. We obtained 1981 ratings. Our Facebook application asks only for *public profiles* and *friend list access* permissions. Also, we anonymize the dataset by mapping Facebook IDs to a random 5-digit number. The study is conducted in two phases: *User Collection Phase* and *Quality Assessment*.

Summary of Results: In summary, we observe that including temporal affinity in group recommendation significantly improves user satisfaction. The amount of satisfaction is variable and is dependent on how groups are formed. In particular, dissimilar user groups as well as those with low-affinity users whose preference significantly evolves over time, are most satisfied. We found that prior work has indeed shown [20] that reaching consensus among such group members is indeed difficult. In addition, we found that **PD**, in general, is the method of choice and works best for dissimilar and high affinity groups. This observation is also in line with one of our prior results [22] where we showed that including disagreement in group consensus generates higher quality recommendations. We also observe that incorporating time models produces better results for high affinity groups suggesting that groups with high affinity are most sensitive to temporal affinities. Finally, the continuous time model is preferred by large groups of dissimilar members. That could be explained because it better captures variability for groups whose members are more sensitive to differences between them. The discrete model on the other hand, is a good approximation of the continuous one in the case of high affinity and high similarity groups.

4.1.1 User Collection Phase

In this phase, the goal is to recruit users and collect their data. Later, collected users are used to form different groups and perform judgments on group recommendations. For this aim, we start with 13 seed users (denoted S). Users in S have to complete two tasks: i. rate at least 30 movies in MovieLens, and ii. invite between 10 and 20 of their friends to participate in the study. The set of friends of a seed user $s \in S$ is denoted $friends(s)$. Note that we consider $\cup_{s \in S} friends(s) \cap S = \emptyset$. Friends are only asked to rate movies and not invite friends, i.e., we stop at the depth 1 of the social graph for this study.

We select a subset of MovieLens movies for participants to provide their preferences. We consider two factors in selecting those movies: *familiarity* and *diversity*. On one hand, we want to present users with a set of movies that they do know about and therefore can provide ratings for. On the other hand, we want to maximize our chances of capturing different tastes among movie-goers. Towards those two goals, we select two sets of movies. The first set is called the *popular set*, which contains the top-50 movies in MovieLens in term of popularity (i.e. the number of users who rated a movie in the set). The second set is called *diversity set*, which contains the 25 movies with the highest variance among their ratings and that are ranked in the top-200 in terms of popularity. Each participant rates movies in one of two pre-computed sets: the **Similar Set** which consists entirely of movies within the *popular set* and the **Dissimilar Set** which consists of the top-25 movies from the *popular set* and the 25 movies from the *diversity set*.

Users are instructed to provide a rating between 1 and 5 (5 being the best) for at least 30 movies listed in random order, according to their preferences.

4.1.2 Static and Dynamic Affinities

In addition to ratings, we store anonymized lists of *friends* and *page-likes* for each user. Since Facebook friendship is relatively stable over time, we use it to compute static affinity: $\mathit{aff}_S(u, u') = |friends(u) \cap friends(u')|$. We normalize all static affinity values in a group by the maximum pair-wise value in the group to obtain a number between 0 and 1.

Page likes are dynamic and are used to compute the time-varying component of affinity. To calculate dynamic affinity for each user, we store all pages (s)he has ever liked in Facebook and for each page, we record the timestamp of when the user liked it and the page category (music, movie, etc.). There exist 197 different page categories in Facebook. For privacy reasons, we do not record the name of the liked pages. Thus the periodic affinity between two users u and u' in time-period p is calculated as: $\mathit{aff}^P(u, u', p) = |page\_likes(u, p) \cap page\_likes(u', p)|$ where $page\_likes(u, p)$ is the set of page categories whose pages are liked by u in time-period p. Then we calculate $\mathit{aff}_V(u, u', p)$ using Equation 1. We also normalize dynamic affinity values to be between 0 and 1. We consider 6 different two-month consecutive periods (Section 4.2.1). Note that the average standard deviation over number of common page-likes for all user pairs during 6 periods is 0.42.

4.1.3 Group Formation

We consider three main factors in forming user groups, i.e., *group size*, *group cohesiveness* and *affinity strength*. Size and cohesiveness (i.e. how similar are group members in their movie tastes) are akin to prior work [22].

We hypothesize that varying group sizes will influence reaching consensus among the members and therefore to which degree members are satisfied with the group recommendation. We choose two group sizes, 3 and 6, representing *small* and *large* groups, respectively.

Similarly, we assume that group cohesiveness is also a significant factor in their satisfaction with group recommendation. As a result, we form two kinds of groups: *similar* and *dissimilar*. A similar group is formed by selecting users who i. have watched **Similar** movies and ii. have the maximum summation of pair-wise similarities (between group members based on their provided ratings) among all groups of the same size. A dissimilar group is formed by selecting users who i. have completed the **Dissimilar** movie set and ii. have the minimum summation of pair-wise similarities among all groups of the same size.

Finally, we consider groups with *low* and *high* affinity between members. We set affinity to be high if each pair-wise affinity in a group is equal to 0.4 or higher.

4.1.4 Quality Assessment

In the second phase of the study, users are instructed to decide which of the recommended movies they are satisfied with in a group. We form 8 groups out of Facebook users by considering different combinations of *group size*, *group cohesiveness* and *affinity strength*. Each user evaluates movies in two phases: *Independent* and *Comparative*.

Independent Evaluation: In the independent evaluation, a user, who is a member of a group, observes a single recommendation list at each time and is asked to say how satisfied she is with watching those movies with other group members using a scale between 0 and 5 (5 being the best). Figure 1 illustrates the results of this

[3] *https://developers.facebook.com*

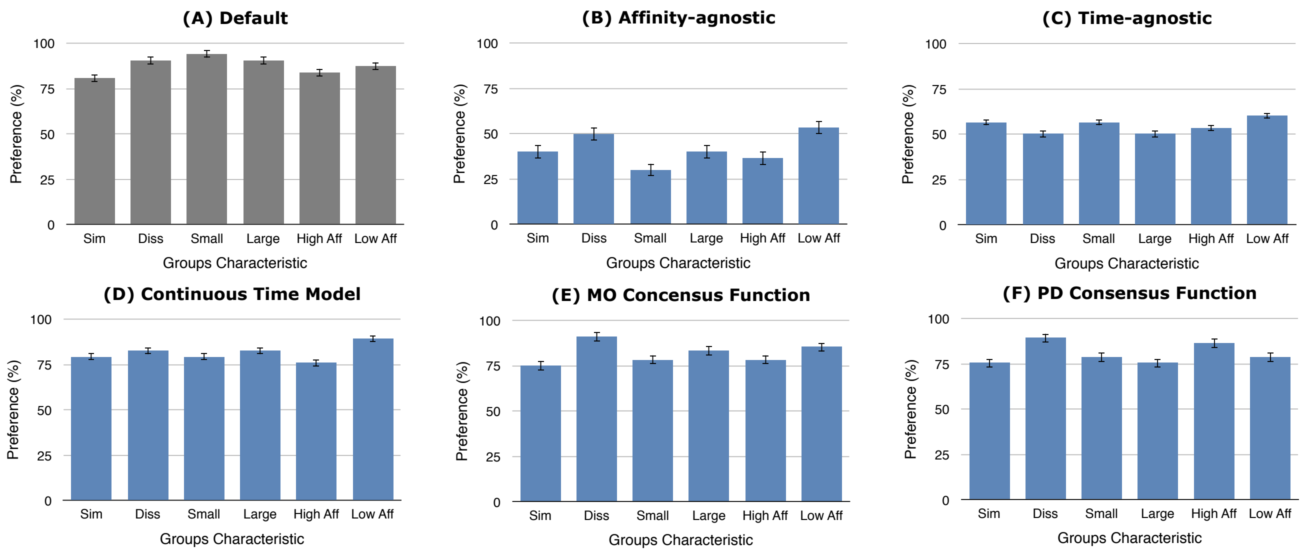


Figure 1: Independent Evaluation

evaluation phase. The score is reported as a percentage, i.e., a result with an average score of 5 gets 100%. Four parameters play a role in generating different recommendation lists in Figure 1, i.e., affinity awareness, time model (discrete vs. continuous), temporal awareness and consensus function. Figure 1. A illustrates results with default values, i.e., affinity-aware, discrete, time-aware and **AP** consensus function. In all other figures, only one parameter value changes, i.e., affinity-agnostic in B, time-agnostic in C, continuous time model in D, **MO** function in E and finally **PD** function in F. That parameter is mentioned in the title of each chart in Figure 1.

We observe that in general, participants give a score of at least 80% to A, which is the default case with discrete temporal affinity. Participants in dissimilar groups have scored A with 90.66% preference while it is 10% lower for similar groups. This could be interpreted as: averaging individual ratings and using a discrete time model works well for groups formed by users who like different movies. Interestingly, the same result holds for low affinity and high affinity groups. This potentially shows that our model is robust to time-varying tastes. On the other hand, the low preference of high affinity groups show that members of those groups benefit from another consensus function, i.e., **PD** (chart F).

Lists without affinity (chart B) and time awareness (chart C) have at most 55% and 60% overall preference respectively. This margin of 20% difference in preference with the temporal affinity case (chart A) shows explicitly the importance of affinity and temporal affinity in group recommendation. In B, worst results are obtained for small (30.08%), high affinity (36.66%) and similar groups (40%) where we observe a decrease in satisfaction. This potentially shows those are the groups that would best benefit from using affinity in computing their recommendations. In C the worst results are for dissimilar and large groups (both 50.19%). One explanation is that dissimilar large groups, i.e., those who differ in their movie tastes among many members, prefer temporal recommendations, i.e., movies that are generated by taking into account their friendship and page-like differences over time.

Groups with different tastes (dissimilar, large and low affinity) prefer the continuous time model (chart D). This is potentially because of a higher precision in capturing time. Considering the time as a whole from the beginning of time is needed to deliver recommendations that satisfy all members of those heterogeneous groups. In case of **MO** (chart E), we observe a superior satisfaction for dissimilar and low affinity groups as *increase in uncertainty in large groups* leads members to like **MO** better.

Comparative Evaluation: In the comparative evaluation, users are asked to compare two lists l_1 and l_2 at a time and pick the list they prefer. Following the closed world assumption, when a list is not chosen by a user, it means that it is not preferred. A user has to choose one and only one of the proposed lists. Figure 3 illustrates the preferences of l_1 over l_2.

First, a user is asked to compare affinity-aware (l_1) vs. affinity-agnostic (l_2) recommendations. In A, we observe that in general, in 75% of the cases, affinity-aware recommendations are preferred. They are mostly appreciated by small groups followed by high affinity groups. Larger groups have less preference for affinity-aware results. A large group potentially leads to higher variability of preference and weaker affinity among its members, thus naturally prohibiting an early agreement.

In the second comparative study, we examine the effect of temporal affinity by comparing time-aware (l_1) vs. time-agnostic (l_2) recommendations. In B, we observe that in most groups, temporal recommendations are preferred in over 80% of the cases. This leaves no doubt that participants like better results obtained based on time. It also shows that high affinity groups prefer not only affinity-based results, but also its temporal version. Small groups have also exhibited a high preference. This is because in groups with fewer members or groups whose participants deeply know each other, the effect of time manifests itself more strongly. Finally, high preference for large groups show that the temporal dimension of affinity is a useful component for such groups to obtain higher quality results, because group members potentially observe that their common affinity history plays a role in recommendations.

We now examine which of the discrete or the continuous temporal affinity models is better and in which case. In C, we observe that in general, the discrete time model is preferred for groups with strong connections between members (high affinity and high similarity). In the case of dissimilar and large groups, it is the continuous model that is preferred. The continuous nature of the latter is certainly better to capture variability for groups whose members are more sensitive to differences between them while the discrete one is a good approximation of the continuous model in the case of high affinity and high similarity groups.

Finally, we compare different group consensus functions. This time, we compare 3 different lists together which are results of **AP**, **MO** and **PD** consensus functions. We are interested to discover which function delivers more satisfactory results when we account for temporal affinities. Figure 2 illustrates this comparison. In short, while the choice of which consensus function to apply heavily depends on group characteristics, there exists a general preference for **PD** especially in the case of loosely connected groups (low affinity and dissimilar groups). That could be explained by the fact that **PD** favors items that minimize disagreement between group members which is more appropriate for dissimilar group members.

In summary, it is shown that **AP** is highly preferred in small and high affinity groups. Whenever **AP** has a high preference, **PD** is also highly preferred. **MO** provides higher quality results for larger groups (this is consistent with our findings in [22]) and for groups with loose connections.

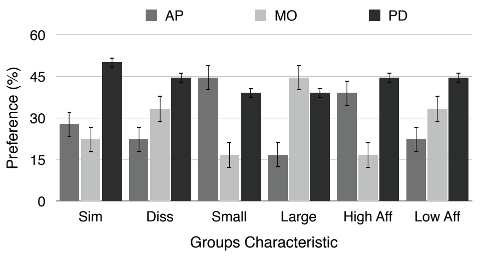


Figure 2: Qualitative Evaluation of Consensus Functions

4.2 Scalability Experiment

Experiment Settings: Unless otherwise stated, we form 20 different random groups by selecting a subset of users who participated in our quality experiment. The default settings of the rest of the parameters are, group size = 6, k = 10, number of items = 3900, consensus function = **AP**. Unless otherwise stated, affinity is computed using the discrete time model. For each scalability experiment, we compute the average percentage of `SA`s needed by `GRECA` in different settings. The percentage of `SA`s represents the computational cost that `GRECA` incurs, compared to a naive algorithm which entirely scans all lists. A smaller percentage exhibits higher scalability.

We conduct experiments by varying time periods, result size (k), group size, number of input items, similarity and dissimilarity among items and users in the group, and considering the discrete and continuous affinity models. Our results illustrate the scalability of `GRECA` with different group consensus functions. We only present a subset of these results. The omitted results are similar to the ones presented. All results are presented with standard error bars, wherever applicable.

Summary of Results: First and foremost, we observe that `GRECA` is highly scalable with varying k, group size, number of items and enables a significant saveup in the number of accesses (almost always, more than 75% accesses are avoided) with early termination. Then, we observe that the pruning ability is highest for similar user groups. We observe that the score distribution of top-k itemsets for such groups is different from the rest of items, therefore, the stopping condition in the buffer is satisfied early. Third, we observe that `GRECA` is effective across all group consensus functions. In fact, for some of the complex group consensus functions that consider user disagreement, `GRECA` incurs the smallest percentage of accesses ensuring the highest saveup in computation cost. Fourth, `GRECA` scales linearly with an increasing number of periods. Finally, we observe that `GRECA` is effective both for discrete and continuous models.

4.2.1 *Varying Time Period*

We explore discretizing time into periods of different lengths: week, month, two-month, season and half-year. Since dynamic affinity relies on user page-likes in Facebook and liking a page is not a frequent action, many time segments were empty after discretization (Figure 4). The length of a time period should be chosen in such a way that each period contains enough data to compute affinities. Figure 4 shows that two-month periods achieve a good balance between the percentage of non-emptiness (65%) and the number of periods (6). We hence pick a two-month discretization for the rest of our experiments.

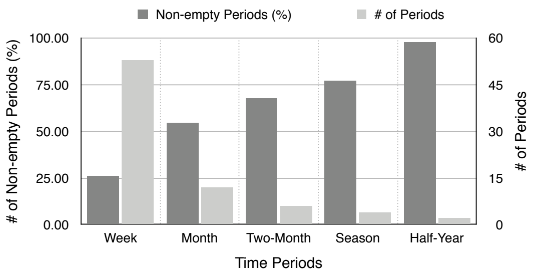


Figure 4: Different Time Periods

Figure 6 illustrates the average number of accesses in each period. As expected, this figure shows a linear behavior in general, as going to subsequent periods increases the number of lists. An exception happens in period 5 where its average #`SA` is very close to its next period. By looking more carefully at the underlying data distribution, we noticed that the number of common page-likes between user pairs in period 5 is very low. Therefore, scanning this period does not help to update bounds in order to have early termination.

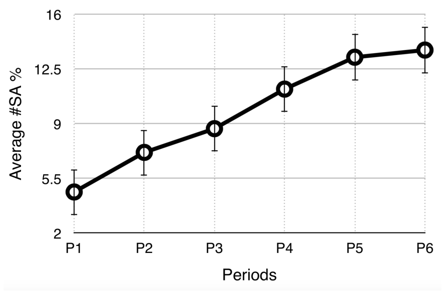


Figure 6: Average Percentage of `SA`s for Different Periods in Discrete Time Model

4.2.2 *Varying k, Group Size and Number of Items*

In Figure 5, we illustrate the scalability of `GRECA` by varying result size, group size and number of items. In A, we vary k from 5 to 30 and run `GRECA` with the **AP** consensus function for 20 different groups with 6 members. We observe that `GRECA` scales linearly with varying k. The algorithm always produces a saveup of 81% or higher.

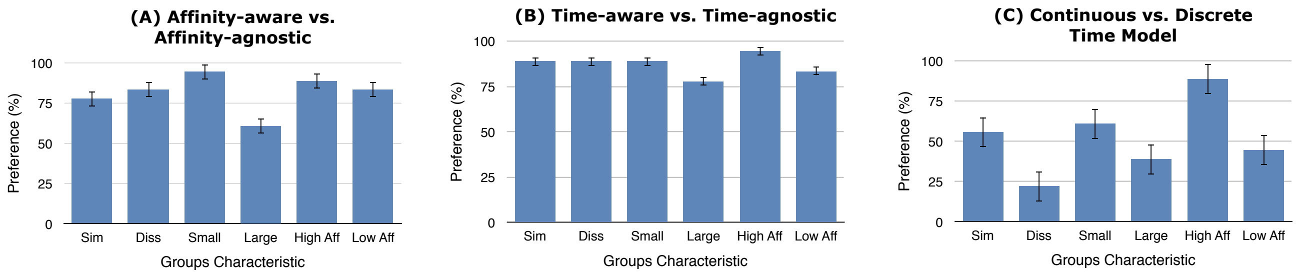


Figure 3: Comparative Evaluation

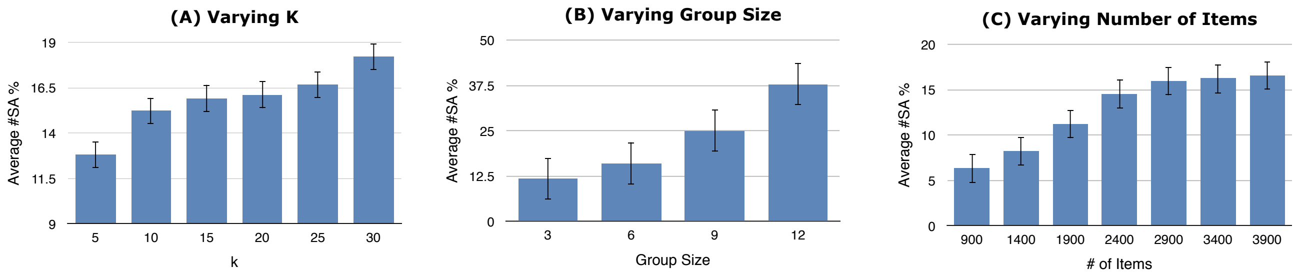


Figure 5: Average Percentage of SAs by Varying Result Size, Group Size and Number of Items

In B, we examine the effect of different group sizes on performance. The results clearly demonstrate that GRECA scales well with varying group sizes. The average saveup is greater than 77%.

In C, we vary the number of available items for group recommendation from 900 to 3900. The results demonstrate that the number of accesses does not necessarily increase with that. This observation is unsurprising as the number of accesses depends on the score distribution of the item preferences and user affinities. GRECA saves more than 83% accesses even in the worst case.

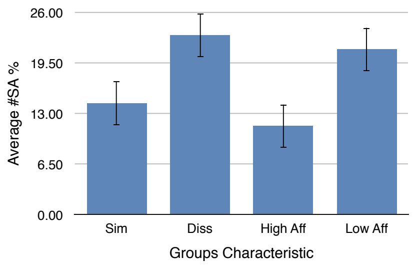


Figure 7: Average Percentage of SAs for Similar, Dissimilar, High Affinity and Low Affinity Groups

4.2.3 *Similarity/Dissimilarity*

We examine the effect of similarity on GRECA in two ways: first, we compare the number of accesses between groups with similar and dissimilar ratings; then, we compare groups with high and low affinities. Figure 7 contains the result. The results demonstrate that the effectiveness is higher for similar groups in both cases (item based similarity and high affinity).

4.2.4 *Time Models*

We examine the effect of continuous and discrete time models on GRECA. The average number of SAs for the continuous model is 16.32% and 16.6% for the discrete one. This means that in both cases, we obtain a saveup greater than 83%. The number of accesses for both methods are very similar with a slight superiority for the discrete model.

4.2.5 *Consensus Functions*

In this last performance study, we compare different consensus functions. Figure 8 contains the results. We introduce two different versions of **PD** based on [22] by varying the weights used in the linear combination of rating aggregation (w_1) and disagreement (w_2) s.t. $w_1 + w_2 = 1$. In **PD V1**, we consider $w_1 = 0.8$ and in **PD V2**, $w_1 = 0.2$.

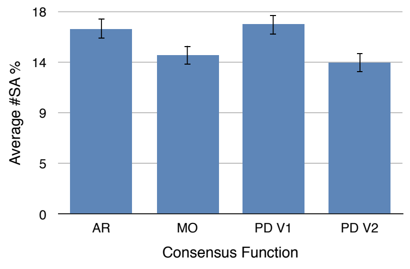


Figure 8: Average Percentage of SAs for Different Consensus Functions

All results clearly demonstrate that GRECA achieves significant saveups for those consensus functions. They also show that **PD V2** outperforms **PD V1**. During our post-analysis, we observed that a higher weight on disagreement allows faster stopping, because the items have smaller scores. **MO** is the next best performer achieving as high as 83% in accesses' saveups.

5. RELATED WORK

Group recommendation has been designed for various domains such as news pages [21], tourism [11] and music [7]. A group may be formed at any time by a random set of users with different interests, a number of persons who explicitly choose to be part of a group, or by computing similarities between users with respect to some similarity functions and then clustering similar users together [19, 3].

There are two dominant strategies for group recommendations [4, 3]. The first approach creates a pseudo-user representing the group and then makes recommendations to that pseudo-user, while the second strategy computes a recommendation list for each group member and then combines them to produce a group's list. For the latter, a widely adopted approach is to apply an aggregation function to obtain a *consensus group preference* for a candidate item. However, to the best of our knowledge, none of the existing functions account for the influence between group members [24].

Affinities may be strong emotional bonds, like links between family members or a clique of close friends. Those links may also be relatively weak thereby breaking with the passage of time or the occurrence of relationship-damaging events. In [16], an affinity model is proposed for group recommendation based on NEO-FFI [4] personality test. Another model is proposed in [12] where 3 different components (social relationship, expertise and disagreement) are aggregated to form affinity. A possible extension of our work could make use of that affinity definition.

On e-commerce platforms, recent studies have proved the importance of time in recommender systems. In [8], Yi Ding et al. assign different weights to different rating records based on their creation time, and reveal the existence of a dynamic change in user interests. Liang Xiong et al. [25] improved the accuracy of recommendations by incorporating the global evolution pattern of user preferences. Potentially the most similar contributions to ours on time models are [17, 15] where Yehuda Koren et al. take temporal dynamics of user and item biases into consideration for individual recommendations. To the best of our knowledge, no previous work has studied a time model for group recommendations.

Threshold algorithms [9] have been used extensively for recommendation. Their attractiveness lies in monotonic score aggregation functions, which operate on sorted input and enable the early pruning of low-ranked answers. In this work, we adapt NRA to aggregate individual preferences, disagreement and temporal affinity lists to compute temporal recommendations and propose a novel stopping condition and prove correctness and instance optimality.

6. CONCLUSION

We examined affinity-aware group recommendation over time and developed GRECA, an efficient algorithm with unique features that distinguish it from state-of-the-art recommendation algorithms. Our proposed semantics is compatible with popular group consensus functions. Our extensive experiments with real Facebook users and Movielens datasets assess the high quality of temporal affinity-aware recommendations for groups with different characteristics (small/large groups, similar/dissimilar groups, high and low affinity groups). In the future we would like to study the maintenance of our index structures over time in relationship with how often affinity between users changes. In particular, we are examining how to combine incremental clustering with our indices in order to determine the minimum amount of information to store that guarantees instance optimality. Moreover, we plan to extend our performance studies to larger groups with thousands of users.

[4] *Neuroticism Extroversion Openness Five Factor Inventory*

References

[1] G. Adomavicius, R. Sankaranarayanan, S. Sen, and A. Tuzhilin. Incorporating contextual information in recommender systems using a multidimensional approach. *ACM Trans. Inf. Syst.*, Jan. 2005.

[2] G. Adomavicius and A. Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *TKDE*, 17(6):734–749, June 2005.

[3] S. Amer-Yahia, S. B. Roy, A. Chawla, G. Das, and C. Yu. Group recommendation: Semantics and efficiency. *PVLDB*, 2(1):754–765, 2009.

[4] S. Berkovsky and J. Freyne. Group-based recipe recommendations: analysis of data aggregation strategies. In *RecSys*, pages 111–118, 2010.

[5] J. R. Bettman, E. J. Johnson, and J. W. Payne. Consumer decision making. *Handbook of consumer behavior*, 44(2):50–84, 1991.

[6] L. Boratto, S. Carta, A. Chessa, M. Agelli, and M. L. Clemente. Group recommendation with automatic identification of users communities. In *Web Intelligence/IAT Workshops*, pages 547–550, 2009.

[7] A. Crossen, J. Budzik, and K. J. Hammond. Flytrap: intelligent group music recommendation. In *IUI*, pages 184–185, 2002.

[8] Y. Ding and X. Li. Time weight collaborative filtering. In *CIKM*, pages 485–492. ACM, 2005.

[9] R. Fagin. Combining fuzzy information: an overview. *SIGMOD Record*, 31(2):109–118, 2002.

[10] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In P. Buneman, editor, *PODS*. ACM, 2001.

[11] I. Garcia, L. Sebastia, and E. Onaindia. On the design of individual and group recommender systems for tourism. *Expert Syst. Appl.*, 38(6):7683–7692, 2011.

[12] M. Gartrell, X. Xing, Q. Lv, A. Beach, R. Han, S. Mishra, and K. Seada. Enhancing group recommendation by incorporating social relationship interactions. In *GROUP*, pages 97–106. ACM, 2010.

[13] A. Jameson and B. Smyth. Recommendation to groups. In *The adaptive web*, pages 596–627. Springer, 2007.

[14] N. M. Klein and M. Yadav. Context effects on effort and accuracy in choice: An inquiry into adaptive decision making. *Journal of Consumer Research*, 16:410–420, 1989.

[15] N. Koenigstein, G. Dror, and Y. Koren. Yahoo! music recommendations: modeling music ratings with temporal dynamics and item taxonomy. In *RecSys*, pages 165–172, 2011.

[16] M. Kompan and M. Bieliková. Social structure and personality enhanced group recommendation. In *EMPIRE 2014, Aalborg, Denmark*, pages 1613–0073, 2014.

[17] Y. Koren. Collaborative filtering with temporal dynamics. *Commun. ACM*, 53(4):89–97, 2010.

[18] D. A. Lussier and R. W. Olshavsky. Task complexity and contingent processing in brand choice. *Journal of Consumer Research*, 6:154–165, 1979.

[19] E. Ntoutsi, K. Stefanidis, K. Nørvåg, and H.-P. Kriegel. Fast group recommendations by applying user clustering. In *ER*, pages 126–140, 2012.

[20] M. O'Connor, D. Cosley, J. A. Konstan, and J. Riedl. Polylens: a recommender system for groups of users. In *ECSCW*, 2001.

[21] S. Pizzutilo, B. De Carolis, G. Cozzolongo, and F. Ambruoso. Group modeling in a public space: Methods, techniques, experiences. AIC'05. WSEAS, 2005.

[22] S. B. Roy, S. Amer-Yahia, A. Chawla, G. Das, and C. Yu. Space efficiency in group recommendation. *VLDB J.*, 19(6):877–900, 2010.

[23] S. B. Roy, S. Thirumuruganathan, S. Amer-Yahia, G. Das, and C. Yu. Exploiting group recommendation functions for flexible preferences. In *ICDE Conference*, 2014.

[24] N. Shabib, J. A. Gulla, and J. Krogstie. On the intrinsic challenges of group recommendation. In *RSWeb@RecSys*, 2013.

[25] L. Xiong, X. Chen, T.-K. Huang, J. G. Schneider, and J. G. Carbonell. Temporal collaborative filtering with bayesian probabilistic tensor factorization. In *SDM*, pages 211–222. SIAM, 2010.

On Processing Top-k Spatio-Textual Preference Queries

George Tsatsanifos
Knowledge and Database Systems Laboratory
National Technical University of Athens
gtsat@dblab.ntua.ece.gr

Akrivi Vlachou
Institute for the Management of Information Systems R.C. "Athena"
avlachou@imis.athena-innovation.gr

ABSTRACT

In this paper we propose a novel query type, termed *top-k spatio-textual preference query*, that retrieves a set of spatio-textual objects ranked by the goodness of the facilities in their neighborhood. Consider for example, a tourist that looks for "hotels that have nearby a highly rated Italian restaurant that serves pizza". The proposed query type takes into account not only the spatial location and textual description of spatio-textual objects (such as hotels and restaurants), but also additional information such as ratings that describe their quality. Moreover, spatio-textual objects (i.e., hotels) are ranked based on the features of facilities (i.e., restaurants) in their neighborhood. Computing the score of each data object based on the facilities in its neighborhood is costly. To address this limitation, we propose an appropriate indexing technique and develop an efficient algorithm for processing our novel query. Moreover, we extend our algorithm for processing spatio-textual preference queries based on alternative score definitions under a unified framework. Last but not least, we conduct extensive experiments for evaluating the performance of our methods.

1. INTRODUCTION

An increasing number of applications support location-based queries, which retrieve the most interesting spatial objects based on their geographic location. Recently, spatio-textual queries have lavished much attention, as such queries combine location-based retrieval with textual information that describes the spatial objects. Most of the existing queries only focus on retrieving objects that satisfy a spatial constraint ranked by their spatio-textual similarity to the query point. However, in addition users are quite often interested in spatial objects (*data objects*) based on the quality of other facilities (*feature objects*) that are located in their vicinity. Feature objects are typically described by *non-spatial* attributes such as quality or rating, in addition to the *textual description*. In this paper, we propose a novel and more expressive query type than existing spatio-textual queries, called *top-k spatio-textual preference query*, for ranked retrieval of data objects based the textual relevance and the non-spatial score of feature objects in their neighborhood.

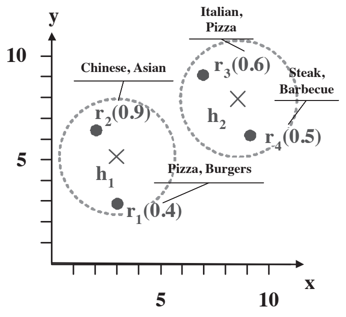


Figure 1: Motivating example.

Consider for example, a tourist that looks for *"hotels that have nearby a highly rated **Italian** restaurant that serves **pizza**"*. Figure 1 depicts a spatial area containing hotels (data objects) and restaurants (feature objects). The quality of the restaurants based on existing reviews is depicted next to the restaurant. Each restaurant also has textual information in the form of keywords extracted from its menu, such as pizza or steak, which describes additional characteristics of the restaurant. The tourist also specifies a spatial constraint (in the figure depicted as a range around each hotel) to restrict the distance of the restaurant to the hotel. Obviously, the hotel h_2 is the best option for a tourist that poses the aforementioned query. In the general case, more than one type of feature objects may exist in order to support queries such as *"hotels that have nearby a good **Italian** restaurant that serves **pizza** and a cheap coffeehouse that serves **muffins**"*. Even though spatial preference queries have been studied before [16, 17, 14], their definition ignores the available textual information. In our example, the spatial preference query would correspond to a tourist that searches for *"hotels that are nearby a good restaurant"* and the hotel h_1 would always be retrieved, irrespective of the textual information.

In this paper, we define top-k spatio-textual preference queries and provide efficient algorithms for processing this novel query type. A main challenge compared to traditional spatial preference queries [16, 17, 14] is that the score of a data object changes depending on the query keywords, which renders techniques that rely on materialization (such

10.5441/002/edbt.2015.38

as [14]) not applicable. Most importantly, processing spatial preference queries is costly in terms of both I/O and execution time [16, 17]. Thus, extending spatial preference queries for supporting also textual information is challenging, since the new query type is more demanding due to the additional textual descriptions.

A straightforward algorithm for processing spatio-textual preference queries is to compute the *spatio-textual preference score* for each data object and then report the k data objects with the highest score. We call this approach *Spatio-Textual Data Scan (STDS)* and examine it as a baseline, while our main focus is to reduce the cost required for computing the spatio-textual score of a data object.

Moreover, we develop an efficient and scalable algorithm, called *Spatio-Textual Preference Search (STPS)*, for processing spatio-textual preference queries. *STPS* follows a different strategy than *STDS*, as it retrieves highly ranked feature objects first, and then searches data objects in their spatial neighborhood. Intuitively, data objects located in the neighborhood of highly ranked feature objects are good candidates for inclusion in the top-k result set. The main challenge tackled with *STPS* is determining efficiently the best feature objects from all feature sets that do not violate the spatial constraint.

To further improve the performance of our algorithms, we develop an appropriate indexing technique called *SRT-index*, that not only indexes the spatial location, the textual description and the non-spatial score, but in addition takes them equally into consideration during the index creation. Finally, we extend our algorithm for processing spatio-textual preference queries based on alternative score definitions under a unified framework. To summarize the contribution of this paper are:

- We propose a novel query type, called top-k spatio-textual preference query, that ranks the data objects based on the quality and textual relevance of facilities (*feature objects*) located in their vicinity.
- A novel indexing technique called *SRT-index* is presented that is beneficial for processing spatio-textual preference queries.
- We present two algorithms for processing spatio-textual preference queries, namely *Spatio-Textual Data Scan (STDS)* and *Spatio-Textual Preference Search (STPS)*.
- We extend our algorithm *STPS* for processing spatio-textual preference queries based on alternative score definitions under a unified framework.
- We conduct an extensive experiment evaluation for studying the performance of our proposed algorithms and indexing technique.

The rest of this paper is organized as follows: Section 2 overviews the relevant literature. In Section 3, we define the spatio-textual preference query. Our novel indexing technique (*SRT-index*) is presented in Section 4. In Section 5 we describe our baseline algorithm, called spatio-textual data scan (*STDS*). An efficient algorithm, called Spatio-Textual Preference Search (*STPS*), is proposed in Section 6. Moreover, we extend our algorithms for processing spatio-textual preference queries based on alternative scores in Section 7. We present the experimental evaluation in Section 8 and we conclude in Section 9.

2. RELATED WORK

Recently several approaches have been proposed for spatial-keyword search. In [8], the problem of distance-first top-k spatial keyword search is studied. To this end, the authors propose an indexing structure (IR^2-Tree) that is a combination of an R-Tree and signature files. The IR-Tree was proposed in another conspicuous work [6, 11], which is a spatio-textual indexing approach that employs a hybrid index that augments the nodes of an R-Tree with inverted indices. The inverted index at each node refers to a pseudo-document that represents all the objects under the node. During query processing, the index is exploited to retrieve the top-k data objects, defined as the k objects that have the highest spatio-textual similarity to a given data location and a set of keywords. Moreover, in [13] the *Spatial Inverted Index (S2I)* was proposed for processing top-k spatial keyword queries. The S2I index maps each keyword to a distinct aggregated R-Tree or to a block file that stores the objects with the given term. All these approaches focus on ranking the data objects based on their spatio-textual similarity to a query point and some keywords. This is different from our work, which ranks the data objects based on textual relevance and a non-spatial score (quality) of the facilities in their spatial neighborhood. [5] provides an all-around evaluation of spatio-textual indices and reports on the findings obtained when applying a benchmark to the indices.

Spatio-textual similarity joins were studied in [1]. Given two data sets, the query retrieves all pairs of objects that have spatial distance smaller than a given value and at the same time a textual similarity that is larger than a given value. This differs from the top-k spatio-textual preferences query, because the spatio-textual similarity join does not rank the data objects and some data objects may appear more than once in the result set. Prestige-based spatio-textual retrieval was studied in [2]. The proposed query takes into account both location proximity and prestige-based text relevance.

The m-closest keywords query [18] aims to find the spatially closest data objects that match with the query keywords. The authors in [3] study the spatial group keyword query that retrieves a group of data objects such that all query keywords appear in at least one data object textual description and such that objects are nearest to the query location and have the lowest inter-object distances. These approaches focus on finding a set of data objects that are close to each other and relevant to a given query, whereas in this paper we rank the data objects based on the facilities in their spatial neighborhood. In [4], the length-constrained maximum-sum region (LCMSR) query is proposed that returns a spatial-network region of constrained size that is located within a general region of interest and that best matches query keywords.

Ranking of data objects based on their spatial neighborhood without supporting keywords has been studied in [15, 7, 16, 17, 14]. Xia *et al.* studied the problem of retrieving the top-k most influential spatial objects [15], where the score of a data object p is defined as the sum of the scores of all feature objects that have p as their nearest neighbor. Yang *et al.* studied the problem of finding an optimal location [7], which does not use candidate data objects but instead searches the space. Yiu *et al.* first considered computing the score of a data object p based on feature objects in its spatial neighborhood from multiple feature sets [16, 17] and defined top-k spatial preference queries. In another line of work, a materialization technique for top-k spatial preference queries was proposed in [14] which leads to significant savings in both computational and I/O cost during query processing. The main difference is that our novel query is defined in addition by a set of keywords that express desirable characteristics of the feature objects (like "pizza" for a feature object that represents a restaurant).

3. PROBLEM STATEMENT

Given an *object* dataset O and a set of c *feature* datasets $\{F_i \mid i \in [1,c]\}$, in this paper, we address the problem of finding k data objects that have in their spatial proximity highly ranked feature objects that are relevant to the given query keywords. Each data object $p \in O$ has a spatial location. Similarly, each feature object $t \in F_i$ is associated with a spatial location but also with a *non-spatial score* $t.s$ that indicates the goodness (quality) of t and its domain of values is the range $[0,1]$. Moreover, t is described by set of keywords $t.\mathcal{W}$ that capture the textual description of the feature object t. Figure 2 depicts an example of a set of feature objects that represent restaurants and shows the non-spatial score and the textual description. Table 1 provides an overview of the symbols used in this paper.

Symbol	Description
O	Set of data objects
p	Data object, $p \in O$
c	Number of feature sets
F_i	Feature sets, $i \in [1,c]$
t	Feature object, $t \in F_i$
$t.s$	Non-spatial score of t
$t.\mathcal{W}$	Set of keywords of t
$dist(p,t)$	Distance between p and t
$sim(t,\mathcal{W})$	Textual similarity between t and $\mathcal{W}$
$s(t)$	Preference score of t
$\tau_i(p)$	Preference score of p based on F_i
$\tau(p)$	Spatio-textual preference score of p

Table 1: Overview of symbols.

The goal is to find data objects that have in their vicinity feature objects that (i) are of high quality and (ii) are relevant to the query keywords posed by the user. Thus, the score of the feature object t captures not only the non-spatial score of the feature, but its textual similarity to a user specified set of query keywords.

Definition 1. *The* ***preference score*** $s(t)$ ***of feature object*** t *based on a user-specified set of keywords* $\mathcal{W}$ *is defined as* $s(t) = (1-\lambda)\cdot t.s + \lambda \cdot sim(t,\mathcal{W})$*, where* $\lambda \in [0,1]$ *and* $sim()$ *is a textual similarity function.*

The textual similarity between the keywords of the feature and the set $\mathcal{W}$ is measured by $sim(t,\mathcal{W})$ and its domain of values is the range $[0,1]$. The parameter λ is the smoothing parameter that determines how much the score of the feature objects should be influenced by the textual information. For the rest of the paper, we assume that the textual similarity is equal to the Jaccard similarity between the keywords of the feature objects and the user-specified keywords: $sim(t,\mathcal{W}) = \frac{|t.\mathcal{W} \bigcap \mathcal{W}|}{|t.\mathcal{W} \bigcup \mathcal{W}|}$.

For example, consider the restaurants depicted in Figure 2. Given a set of keywords $\mathcal{W} = \{italian,\ pizza\}$ and $\lambda = 0.5$ the restaurant with the highest preference score is *Ontario's Pizza* with a preference score $s(r_6) = 0.9$, while the score of *Beijing Restaurant* is $s(r_1) = 0.3$, since none of the given keywords are included in the description of *Beijing Restaurant*.

Given a spatio-textual preference query Q defined by an integer k, a range r and c-sets of keywords $\mathcal{W}_i$, the preference score of a data object $p \in O$ based on a feature set F_i is defined by the scores of feature objects $t \in F_i$ in its spatial neighborhood, whereas the overall spatio-textual score of p is defined by taking into account all feature sets F_i, $1 \le i \le c$.

Definition 2. *The* ***preference score*** $\tau_i(p)$ ***of data object*** p *based on the feature set* F_i *is defined as:* $\tau_i(p) = max\{s(t) \mid t \in F_i : dist(p,t) \le r \text{ and } sim(t,\mathcal{W}_i) > 0\}$.

The $dist(p,t)$ denotes the spatial distance between data object p and feature object t and we employ the Euclidean distance function. Continuing the previous example, Figure 4 shows the spatial location of the restaurants in Figure 2 and a data point p that represents a hotel. The preference score of p based on the restaurants in its neighborhood (assuming $r = 3.5$ and $\mathcal{W} = \{italian,\ pizza\}$) is equal to the score of r_6 ($\tau_i(p) = s(r_6) = 0.9$), which is the best restaurant in the neighborhood of p.

Definition 3. *The overall* ***spatio-textual preference score*** $\tau(p)$ ***of data object*** p *is defined as:* $\tau(p) = \sum_{i\in[1,c]} \tau_i(p)$.

Figure 3 shows a second set of feature objects that represents coffeehouses. For a tourist that looks for a good hotel that has nearby a good Italian restaurant that serves pizza and a good coffeehouse that serves espresso and muffins, the score of p would be $\tau(p) = s(r_6) + s(c_5) = 0.9 + 0.78233 = 1.6833$.

Problem 1. Top-k Spatio-Textual Preference Queries (STPQ)*: Given a query Q, defined by an integer k, a radius r and c-sets of keywords* $\mathcal{W}_i$*, find the k data objects* $p \in O$ *with the highest spatio-textual score* $\tau(p)$.

4. INDEXING

The main difference of top-k spatio-textual preference queries to traditional spatio-textual search is that the ranking of a data object does not depend only on spatial location and textual information, but also on the non-spatial score of the feature object. In particular, the preference score $s(t)$ of feature object t is defined by its textual description and its non-spatial score, while the spatial location is used as a filter for computing the preference score $\tau_i(p)$ of data object p. Thus, efficient indexing of the textual description and the non-spatial score of feature objects is a significant factor for designing efficient algorithms for the STPQ query.

4.1 Index Characteristics

In this paper, we assume that the data objects O are indexed by an R-Tree, denoted as *rtree*. However, for the feature objects, it is important that the non-spatial score and the textual description are indexed additionally. Each dataset F_i can be indexed by any spatio-textual index that relies on a spatial hierarchical index (such as the R-Tree). However, each entry e of the index must in addition maintain: (i) the maximum value of $t.s$ of any feature object t in the sub-tree, denoted as $e.s$, and (ii) a summary ($e.\mathcal{W}$) of

	name	rating	x	y	textual description
r_1	Beijing Restaurant	0.6	1	2	Chinese, Asian
r_2	Daphne's Restaurant	0.5	4	1	Greek, Mediterranean
r_3	Espanol Restaurant	0.8	5	8	Italian, Spanish, European
r_4	Golden Wok	0.8	2	3	Chinese, Buffet
r_5	John's Pizza Plaza	0.9	8	4	Pizza, Sandwiches, Subs
r_6	Ontario's Pizza	0.8	7	6	Pizza, Italian
r_7	Oyster House	0.8	6	10	Seafood, Mediterranean
r_8	Small Bistro	1.0	3	7	American, Coffee, Tea, Bistro

Figure 2: Feature objects (Restaurants)

	name	rating	x	y	textual description
c_1	Bakery & Cafe	0.6	4	1	Cake, Bread, Pastries
c_2	Coffee House	0.5	4	7	Cappuccino,Toast, Decaf
c_3	Coffe Time	0.8	3	10	Cake, Toast, Donuts
c_4	Cafe Ole	0.6	6	2	Cappuccino, Iced Coffee, Tea
c_5	Royal Coffe Shop	0.9	5	5	Muffins, Croissants,Espresso
c_6	Mocha Coffe House	1.0	10	3	Macchiato, Espresso, Decaf
c_7	The Terrace	0.7	6	9	Muffins, Pastries, Espresso
c_8	Espresso Bar	0.4	7	6	Croissants, Decaf, Tea

Figure 3: Feature objects (Coffeehouses)

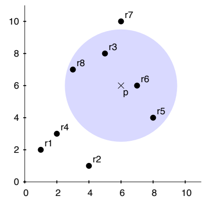


Figure 4: An example of a $STPQ$ query.

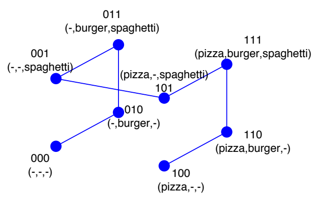


Figure 5: Hilbert-based keyword ordering.

all keywords of any feature t in the sub-tree. To ensure correctness of our algorithms, there must exist an upper bound $\widehat{s}(e)$ such that for any t stored in the sub-tree rooted by the entry e it holds:

$$\widehat{s}(e) \geq s(t)$$

The above property guarantees that the preference score $s(t)$ of a feature object t is bounded by the bound $\widehat{s}(e)$ of its ancestor node e. The efficiency of the algorithms directly depends on the tightness of this bound. In turn, this depends on the similarity between the textual description and the non-spatial score of the features objects that are indexed in the same node.

In the following, we propose an indexing technique that leads to tight bounds since objects with similar textual information and non-spatial score are stored in the same node of the index.

4.2 Indexing based on Hilbert Mapping

Our indexing approach maps the textual description of feature objects to a value based on the Hilbert curve. Let w denote the number of distinct keywords in the vocabulary, then for each feature t the keywords $t.\mathcal{W}$ can be represented as a binary vector of length w. For instance, assuming a vocabulary $\{pizza, burger, spaghetti\}$, we can use an active bit to declare the existence of the *"pizza"* keyword at the first place, *"burger"* at the second, and *"spaghetti"* at the last. Moreover, we suggest a mapping of the binary vector to a Hilbert value, denoted as $\mathcal{H}(t.\mathcal{W})$. For the above w=3 keywords, the defined order is 000,010,011,001,101,111,110 and 100. Figure 5 shows the ordering of the keywords based on the Hilbert values. The benefit of this order is that it ensures us that vectors with distance 1 have only one different keyword, while if the distance is w', then the maximum number of different keywords is bound by w'. This means that consecutive vectors in the afore-described order have only few different keywords, which means that objects with sequential $\mathcal{H}$-values are highly similar also based on the Jaccard similarity function.

Using the Hilbert mapping of the textual information, each feature object t can be represented as a point in the 4-dimensional space $\{t.x,\ t.y,\ t.s,\ \mathcal{H}(t.\mathcal{W})\}$. Our indexing technique, called *SRT-index*, uses a spatial index, such as a traditional R-Tree, that is built on the mapped 4-dimensional space. In terms of structure, the SRT-index resembles a traditional R-Tree that it is built on the spatial location, the non-spatial score (rating), and the Hilbert value of the keywords of the feature objects altogether. The only modification needed during the index construction is the method used for updating the Hilbert values of a node. When the Hilbert value of a node is updated because a new object is added, then the previous Hilbert value as well the Hilbert value of the new object are mapped to binary vectors, the disjunction of the binary vectors is computed, mapped to a new Hilbert value and stored in the node. Notably, the exact spatial index used for indexing the mapped space does not affect the correctness of our algorithms, but only their performance. In our experimental evaluation, we use bulk insertion [9] on our novel indexing technique.

During query processing the bound $\widehat{s}(e)$ of a node e can be set as:

$$\widehat{s}(e) = (1-\lambda)\cdot e.s + \lambda \cdot \frac{|e.\mathcal{W} \bigcap \mathcal{W}|}{|\mathcal{W}|}$$

where $\mathcal{W}$ is the set of query keywords, while $e.\mathcal{W}$ is the set of all keywords of all feature objects t indexed by the node e. The set $e.\mathcal{W}$ is computed based on the Hilbert mapping and the aggregated Hilbert value $\mathcal{H}(e.\mathcal{W})$ stored in the node entry e of the SRT-tree. It holds that $\widehat{s}(e) \geq s(t)$.

To summarize, the SRT-index overcomes the difficulty that other indexing approaches face, being unable to identify in advance what are the branches of the index that store highly ranked and relevant feature objects to the query. The reason is that this indexing mechanism can identify effectively the promising parts of the hierarchical structure at a low cost, since during the index construction the similarity of the spatial location, the non-spatial score, as well as the textual description are taken into account.

5. SPATIO-TEXTUAL DATA SCAN (STDS)

Our baseline approach, called spatio-textual data scan ($STDS$), computes the spatio-textual score $\tau(p)$ of *each* data object $p \in O$ and then reports the k data objects with the highest score. Algorithm 1 shows the pseudocode of $STDS$.

In more detail, for a data object p, its score $\tau_i(p)$ for every feature set F_i is computed (lines 3-5). The details on this computation for range queries are described in Algorithm 2 that will be presented in the sequel. Interestingly, for some data objects p we can avoid computing $\tau_i(p)$ for some feature sets. This is feasible because we can determine early that some data objects cannot be in the result set R. To achieve this goal, we define a threshold τ which is the k-th highest score of any data object processed so far. In addition, we define an upper bound $\widehat{\tau}(p)$ for the spatio-textual preference score $\tau(p)$ of p, which does not require knowledge of the preference scores $\tau_i(p)$ for all feature sets F_i: $\widehat{\tau}(p) = \sum_{i\in[1,c]} \begin{cases} \tau_i(p), & \textit{if } \tau_i(p) \textit{ is known} \\ 1, & \textit{otherwise} \end{cases}$. The algorithm tests the upper bound $\widehat{\tau}$ based on the already computed $\tau_i(p)$ against the current threshold (line 6). If $\widehat{\tau}$ is smaller than the current threshold, the remaining score computations are avoided. After computing the score of p, we test whether it belongs to R (line 6). If this is case, the result set R is updated (line 7), by adding p to it and removing the data object with the lowest score (in case that $|R| > k$). Finally, if at least k data objects have already been added to R, we update the threshold based on the k-th highest score (line 9).

Algorithm 1: *Spatio-Textual Data Scan (*STDS*)*

```
   Input: Query $Q = (k, r, \{\mathcal{W}_i\})$
   Output: Result set $R$ sorted based on $\tau(p)$
1  $R = \emptyset$; $\tau = -1$;
2  foreach $p \in O$ do
3      for $i = 1 \ldots c$ do
4          if $\widehat{\tau}(p) > \tau$ then
5              $\tau_i(p) = F_i.computeScore(Q, p)$ ;
6      if $\tau(p) > \tau$ then
7          $update(R)$ ;
8          if $|R| \geq k$ then
9              $\tau = k^{th}$ score ;
10 return $R$ ;
```

The remaining challenge is to compute efficiently the score based on the spatio-textual information of the feature objects. The goal is to reduce the number of disk accesses for retrieving feature objects that are necessary for computing the score of each element $p \in O$. Algorithm 2 shows the computation of preference score $\tau_i(p)$ for feature set F_i. First, the root entry is retrieved and inserted in a heap (line 1). The heap maintains the entries e sorted based on their values $\widehat{s}(e)$. In each iteration (lines 2-11), the entry e with the highest value $\widehat{s}(e)$ is processed, following a best-first approach. If e is a data point and within distance r from p (line 5), then the score $\tau_i(p)$ of p has been found and is returned (line 7). If e is not a data point, then we expand it only if it satisfies the query constraints (line 9). More detailed, if the minimum distance of e to p is smaller or equal to r and its textual similarity is larger than 0, e is expanded and its child entries are added to the heap (line 11). Otherwise, the entire sub-tree rooted at e can be safely pruned.
Correctness and Efficiency: Algorithm 2 always reports the correct score $\tau_i(p)$. The sorted access of the entries, combined with the property that the value $\widehat{s}(e)$ of the entry is an upper bound ensure its correctness. Moreover, it can be shown that Algorithm 2 expands the minimum number of

Algorithm 2: *Spatio-Textual Score Computation on F_i ($computeScore(Q, p)$)*

```
   Input: Query $Q$, data object $p$
   Output: Score $\tau_i(p)$
1  $heap$.push($F_i$.root);
2  while (not $heap.isEmpty()$) do
3      $e \leftarrow heap$.pop() ;
4      if $e$ is a data object then
5          if ($dist(p, e) \leq r$) then
6              $\tau_i(p) = s(e)$ ;
7              return $\tau_i(p)$ ;
8      else
9          if ($mindist(p, e) \leq r$) and ($sim(e, \mathcal{W}_i) > 0$) then
10             for $childEntry$ in $e.childNodes$ do
11                 $heap$.push(childEntry) ;
```

entries, in the sense that if an entry that is expanded was not expanded, it could lead to computing a wrong score. This is because only entries with score higher than any processed feature object are expanded, and such entries may contain in their sub-tree a feature object with score equal to the score of the entry.
Performance improvements: The performance of *STDS* can be improved by processing the score computations in a batch. Instead of a single data object p, a set of data objects $\mathcal{P}$ can be given as input to Algorithm 2. Then, an entry is expanded if the distance for *at least* one p in $\mathcal{P}$ is smaller than r. When a feature object is retrieved, for any p for which the distance is smaller than r the score is computed and those data objects p are removed from $\mathcal{P}$. The same procedure is followed until either the heap or $\mathcal{P}$ is empty. Algorithm 1 can be easily modified to invoke Algorithm 2 for all data objects in the same leaf entry of the R-tree (*rtree*) that indexes the data objects O. For sake of simplicity, we omit the implementation details, even though we use this improved modification in our experimental evaluation.

6. SPATIO-TEXTUAL PREFERENCE SEARCH (STPS)

In this section we propose a novel and efficient algorithm, called Spatio-Textual Preference Search (*STPS*), for processing spatio-textual preference queries. *STPS* follows a different strategy than *STDS*, as it involves two major steps, namely finding highly ranked feature objects first, and then, retrieving data objects in their spatial neighborhood. Intuitively, if we find a neighborhood in which highly ranked feature objects exist, then the neighboring data objects are naturally highly ranked as well.

6.1 Valid Combination of Feature Objects

In a nutshell, the goal is to find sets of feature objects $\mathcal{C} = \{t_1, t_2, \ldots, t_c\}$ where $t_i \in F_i$ $(1 \leq i \leq c)$, such that the spatio-textual preference score of each t_i is as high as possible and the feature objects are located in nearby locations.

In the general case, a data object may be highly ranked even in the case where a certain kind of feature object does not exist in its neighborhood, though feature objects of other kinds might compensate for this. For example, consider the extreme case where all data objects have only one type of feature object in their spatial neighborhood. For ease of pre-

Algorithm 3: *Spatio-Textual Preference Search (STPS)*

```
  Input: Query $Q$
  Output: Result set $R$ sorted based on $\tau(p)$
1 while $(|R| \leq k)$ do
2     $\mathcal{C} = nextCombination(Q)$ ;
3     $R = R\cup$ getDataObjects($\mathcal{C}$) ;
4 return $R$ ;
```

sentation, we denote as $\emptyset$ a virtual feature object for which it holds that $dist(p,\emptyset) = 0$, $dist(t_i,\emptyset) = 0$ and $s(\emptyset) = 0$ $\forall t_i, p$. This virtual feature object is used for presenting unified definitions for the case where the spatio-textual score of the top-k data objects is defined based on less than c feature objects. More formally put, we define the concept of *valid combination* of feature objects as:

Definition 4. *A valid combination of feature objects is a set $\mathcal{C} = \{t_1, t_2, \ldots, t_c\}$ such that (i) $\forall i$ $t_i \in F_i$ or $t_i = \emptyset$, and (ii) $dist(t_i, t_j) \leq 2r$ $\forall i, j$. The score of the valid combination $\mathcal{C}$ is defined as $s(\mathcal{C}) = \sum_{1\leq i\leq c} s(t_i)$.*

The following lemma proves that it is sufficient to examine only the valid combinations $\mathcal{C}$ of feature objects in order to retrieve the result set of a top-k spatio-textual preference query.

Lemma 1. *The score of any data object $p \in O$ is defined by a valid combination of feature objects $\mathcal{C} = \{t_1, t_2, \ldots, t_c\}$, i.e., $\forall p : \exists \mathcal{C} = \{t_1, t_2, \ldots, t_c\}$ such that $\tau(p) = s(\mathcal{C})$*

Proof. Let us assume that there exists p such that: $\tau(p) = \sum_{i\in[1,c]} \tau_i(p)$ with $\tau_i(p) = \{s(t_i) \mid t_i \in F_i : dist(p,t_i) \leq r \text{ and } sim(t_i, \mathcal{W}_i) > 0\}$ and $\mathcal{C} = \{t_1, t_2, \ldots, t_c\}$ is not a valid combination of feature objects. Since $\mathcal{C} = \{t_1, t_2, \ldots, t_c\}$ is not a valid combination of feature objects, there exists $1 \leq i \neq j \leq c$ such that $dist(t_i, t_j) > 2r$ but also $dist(p, t_i) \leq r$ and $dist(p, t_j) \leq r$. Based on the triangular inequality it holds: $dist(t_i, t_j) \leq dist(p, t_i) + dist(p, t_j) \leq r + r \leq 2r$, which is a contradiction. □

6.2 STPS Overview

Algorithm 3 provides an insight to *STPS* algorithm. At each iteration, the following steps are followed: (i) a special iterator (line 2) returns successively the valid combinations of feature objects sorted based on their score (we discuss the details on the implementation of the iterator in the following subsection), (ii) up to k data points in the spatial neighborhood of these features are retrieved (line 3). Data objects that have already been previously retrieved are discarded, while the remaining data objects p have a score $\tau(p) = s(\mathcal{C})$ and can be returned to the user incrementally. If k data objects have been returned to the user (line 1), the algorithm terminates without retrieving the remaining combinations of feature objects. Differently to the *STDS* algorithm, *STPS* retrieves only the data objects that most certainly belong to the result set.

6.3 Spatio-Textual Feature Objects Retrieval

Algorithm 4 shows the pseudocode for retrieving the valid combinations $\mathcal{C} = \{t_1, t_2, \ldots, t_c\}$ of feature objects sorted based on their spatio-textual preference score $s(\mathcal{C})$. We first give a scketch of our algorithm and then we will elaborate

Algorithm 4: *Spatio-Textual Feature Objects Retrieval (nextCombination(Q))*

```
   Input: Query $Q$
   $heap_i$: heap maintaining entries of $F_i$
   $heap$: heap maintaining valid combinations of feature objects
   $\mathcal{D}_i$: set of feature objects of $F_i$
   Output: $\mathcal{C}$: valid combination with highest score
 1 while $(\exists i :$ not $heap_i.isEmpty())$ do
 2     $i \leftarrow$ nextFeatureSet() ;
 3     $e_i \leftarrow heap_i.pop()$ ;
 4     while (not $e_i$ is a data object) do
 5         for childEntry in $e_i.childNodes$ do
 6             $heap_i$.push(childEntry) ;
 7         $e_i \leftarrow heap_i$.pop() ;
 8     $\mathcal{D}_i = \mathcal{D}_i \cup e_i$ ;
 9     $heap$.push(validCombinations($\mathcal{D}_1, \cdots, e_i, \cdots, \mathcal{D}_c$)) ;
10     $min_i = s(e_i)$ ;
11     $\tau = max_{1\leq j\leq c}(max_1 + \cdots + min_j + \cdots + max_c)$ ;
12     $\mathcal{C} \leftarrow heap$.top() ;
13     if $(score(\mathcal{C}) \geq \tau)$ then
14         $heap$.pop() ;
15         return $\mathcal{C}$;
```

further on the details in the following of this section. In each iteration, a feature set F_i is selected (line 2) based on a pulling strategy implemented by *nextFeatureSet*(). The spatio-textual index that stores the feature objects of the feature set F_i is accessed and the feature objects t_i are retrieved based on their score $s(t_i)$ that aggregates their non-spatial score, but also their textual similarity to the query keywords (lines 3-7). The retrieved feature objects are maintained in a list $\mathcal{D}_i$ (line 8) and are used to produce valid combinations $\mathcal{C}$ of feature objects (line 9). Moreover, a thresholding scheme is employed to decide when the combination with the highest score has been retrieved (lines 11-15).

We denote as max_i the maximum score of $\mathcal{D}_i$ and min_i the minimum score of $\mathcal{D}_i$. Thus, min_i represents the best potential score of any feature object of F_i that has not been processed yet. Moreover, in Algorithm 4 the variables $heap_i$, $\mathcal{D}_i$, max_i, min_i, and $heap$ are global variables. They are initialized as following $heap_i$: the root of F_i, $\mathcal{D}_i = \emptyset$ and $heap = \emptyset$, $min_i = \infty$. Variable max_i is the score of the highest ranked feature object of F_i and is set the first time the F_i index is accessed.

Accessing F_i: In each iteration, Algorithm 4 accesses one spatio-textual index that stores the set F_i (lines 3-7). The entries of the spatio-textual index responsible for the feature objects of F_i are maintained in $heap_i$, which keeps the entries e sorted based on $\widehat{s}(e)$. Moreover, for sake of simplicity, we assume that $heap_i.pop()$ will return a virtual feature object $t_i = \emptyset$ (with score equal to 0) as final object. In each iteration an entry e_i of the spatio-textual index is retrieved from $heap_i$ (line 3). If the entry e_i corresponds to a node of the index, the entry is expanded and its child nodes are added to the $heap_i$ (lines 5-6). Algorithm 4 continues retrieving from $heap_i$ entries, until an entry that is a feature object is retrieved (line 4). When an entry e_i is retrieved that corresponds to a feature object, e_i is inserted in the list $\mathcal{D}_i$ (line 8).

Creation of $\mathcal{C}$: After retrieving a new feature object e_i, new valid combinations $\mathcal{C}$ are created by combining e_i with the previously retrieved feature objects t_j maintained in the lists $\mathcal{D}_j$ (line 9). For this, the method *validCombinations* is called, which returns all combinations of the objects in $\mathcal{D}_j$ and e_i, by discarding combinations for which the condition $dist(t_i, t_j) \leq 2r \ \forall i, j$ does not hold. The new valid combinations are inserted in the *heap* (line 9) that maintains the valid combinations sorted based on their score $s(\mathcal{C})$.

Thresholding scheme: Algorithm 4 employs a thresholding scheme to determine if the current best valid combination can be returned as the valid combination with the highest score. The threshold τ represents the best score of any valid combination of feature objects that has not been examined yet. The best score of the next feature object t_j retrieved from F_j is equal to min_j, since the feature objects are accessed sorted based on $s(t_j)$. Obviously, for the remaining feature sets we assume that the new feature object t_j is combined with the feature objects that have the highest score. Thus, $\tau = max_{1\leq j \leq c}(max_1 + \cdots + min_j + \cdots + max_c)$ (line 11) is an upper bound of the score for any valid combination that has not been examined yet. In line 13, we test whether the best combination of feature objects in the *heap* has a score higher or equal to the threshold τ. If so, the best combination in the heap is the next valid combination with the best score. Otherwise, additional feature objects from feature sets F_i have to be retrieved until it holds that the top element of the *heap* achieves a score which is higher than τ.

Pulling strategy: In the following, we proposed an advanced pulling strategy that prioritizes retrieval from feature sets that have higher potential to produce the next valid combination $\mathcal{C}$. A simple alternative would be to access the different feature sets in a round robin fashion.

The order in which the feature objects of different feature sets are retrieved is defined by a pulling strategy, i.e., *nextFeatureSet*() returns an integer between 1 and c and defines the pulling strategy. In addition, *nextFeatureSet*() never returns i if $heap_i$ is empty.

Definition 5. *Given c sets of feature objects $\mathcal{D}_i$, the prioritized pulling strategy returns m as the next feature set such that $\tau = max_1 + \cdots + min_m + \cdots + max_c$.*

The main idea of the prioritized pulling strategy is that in each iteration the feature set F_m that is responsible for the threshold value τ is accessed. It is obvious that the only way to reduce τ is to reduce the min_m, since retrieval from the remaining feature sets cannot reduce τ. Thus, retrieving the next tuple from the feature set F_m may reduce the threshold τ and may produce new valid combinations that have a score equal to the current threshold.

6.4 Retrieval of Qualified Data Objects

In the following, we study the reciprocal actions taken upon the formation of a highly ranked combination of feature objects.

In Algorithm 3 (line 3) *getObjects($\mathcal{C}$)* is invoked to retrieve from *rtree* all data objects in the neighborhood of the feature objects in $\mathcal{C}$. This method starts from the root of the *rtree* and processes its entries recursively. Entries e for which $\exists i$ such that $t_i \in \mathcal{C}$ with $dist(e, t_i) > r$ are discarded. The remaining entries are expanded until all objects p for which it holds that $dist(p, t_i) \leq r$ are retrieved.

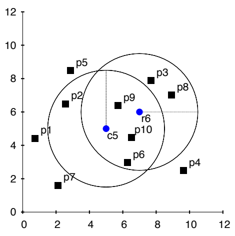


Figure 6: Data objects within qualifying distance from $\mathcal{C} = \{r_6, c_5\}$.

Example. *Consider for example the feature sets depicted in Figure 2 and in Figure 3. Given a query with $r = 3.5$, $\mathcal{W}_1 = \{italian,\ pizza\}$ and $\mathcal{W}_2 = \{espresso,\ muffins\}$, the restaurant and the coffeehouse with the highest scores are r_6 and c_5 respectively. Since it holds that $dist(r_6, c_5) \leq 2r$, the set $\mathcal{C} = \{r_6, c_5\}$ is a valid combination of feature objects. Assume that the set of data objects is $O = \{p_1, p_2, \ldots, p_{10}\}$ as depicted in Figure 6. For the data objects p_6, p_9 and p_{10} it holds that $dist(p_i, c_5) \leq r$ and $dist(p_i, r_6) \leq r$, and their spatial-textual score is $\tau(p_6) = \tau(p_9) = \tau(p_{10}) = 1.6833$. These data objects are guaranteed to be the highest ranked data objects and can be immediately returned to the user. For $k \leq 3$, our algorithm terminates without examining other feature combinations.*

7. VARIANTS OF TOP-K SPATIO-TEXTUAL PREFERENCE QUERIES

In this section, we extend our algorithms for processing spatio-textual preference queries based on alternative score definitions under a unified framework. We provide formal definitions for the alternative score definitions, namely *influence preference score* and *nearest neighbor preference score.* Moreover, we discuss for all query types the necessary modifications to our query processing algorithms.

7.1 Influence-Based STPQ Queries

In contrast to the preference score defined in Definition 1 (in the following referred to as range score), in this section we define an alternative score that does not pose a hard constraint on the distance, but instead gradually reduces the score based on the distance. We call this variant *influence preference score.*

Definition 6. *The **influence preference score** $\tau_i(p)$ of data object p based on the feature set F_i is defined as: $\tau_i(p) = max\{s(t) \cdot 2^{\frac{-dist(p,t)}{r}} \mid t \in F_i :\ sim(t, \mathcal{W}_i) > 0\}$.*

The overall spatio-textual score $\tau(p)$ of data object p is defined as in the case of the range score, and the query returns the k objects with the highest score.

The *STDS* algorithm, as defined in Algorithm 1 can be easily adapted for the case of influence score. Only the function *computeScore*(Q, p) must be modified according to the definition of the score variant. Thus, in Algorithm 2 each entry in line 11 will be prioritized according to the influence preference score. In addition, the range restriction is removed in line 5 and line 9. No further modifications are needed, thus in the following we focus on the modifications and optimizations needed for *STPS* algorithm.

Algorithm 5: *STPS for influence score*

```
Input: Query $Q$
Output: Result set $R$ sorted based on $\tau(p)$
1 $\tau = 0$ ;
2 $score = -1$ ;
3 while ($|R| \leq k$) or ($best > \tau$) do
4     $\mathcal{C} = nextCombination(Q)$ ;
5     $best = s(\mathcal{C})$ ;
6     $R = R\cup$ getDataObjects($\mathcal{C}$) ;
7     $\tau = k$-th score in $R$ ;
8 return $R$ ;
```

STPQ queries based on the influence preference score can be efficiently supported by the *STPS* algorithm with few modifications. Algorithm 5 shows the modified *STPS* for influence preference score. The algorithm continues until at least k data object have been retrieved and until we are sure that none of the remaining data objects can have a better score. We use the score of the k-th data object of the current top-k result (line 7) to set a threshold τ. Hence, if the *best* score of any unseen combination is smaller or equal to τ, the algorithm naturally terminates. In more details, $\mathcal{C} = nextCombination(Q)$ is the same with Algorithm 4 and returns the best combination based on score $s(\mathcal{C})$, but without discarding combinations whose distance is greater than $2r$. Thus, in each iteration the combination $\mathcal{C}$ with the highest $\tau(p) = \sum_{i\in[1,c]} \tau_i(p)$ is retrieved. Recall that for the case of the range preference score, all data objects that were located in distance smaller than r from all feature objects of $\mathcal{C}$ had a score equal to $s(\mathcal{C})$. Instead in the case of the influence preference score, $s(\mathcal{C})$ is an upper bound for the score of all data objects based on $\mathcal{C}$. This is because, the computed score is the influence score only for the objects with distance 0, while all other objects have a smaller influence score. Therefore, *getDataObjects*($\mathcal{C}$) must be modified accordingly.

In more details, *getDataObjects*() retrieves the k points that have the highest influence score, by starting a top-k query on the R-Tree (*rtree*) of the data objects. The root is inserted in a heap sorted by the influence score ($\tau(p) = \sum_{i\in[1,c]} \tau_i(p)\dot{2}^{\frac{-dist(p,t_i)}{r}}$). For non-leaf entries e the influence score is computed based on the mindist. Then, the influence score of an entry is an upper bound of any object in the subtree. After retrieving k data objects, we have retrieved the k data objects with the highest influence score for this combination of feature objects. Further improvement can be achieved if *getDataObjects*() stops retrieving data objects based on τ, which reduces the I/Os on *rtree*. If τ is given to *getDataObjects*() then it will return at most k data objects that have a score smaller than τ. Line 6 merges the results while it removes objects that have been retrieved before. Thus, if an object that is already in the heap is retrieved again the score with the highest value is kept.

After retrieving k data objects with the highest $\tau(p)$ in line 6 (Algorithm 5), the score of the k-th data object in R is used as a threshold τ (line 7). The best score of any unseen combination is $best = s(\mathcal{C})$, which is also an upper bound for the score of any unseen data object, since this is the score for distance 0. Hence, if the *best* score is greater than τ, we have to retrieve additional objects. If the score $s(\mathcal{C})$ of the next combination is smaller than or equal to the threshold we stop retrieving other combinations.

7.2 Nearest Neighbor STPQ Queries

In the next score variant, each data object takes as a score the goodness of the feature objects that are its nearest neighbors. In particular, for each feature set the score of the nearest feature object is considered for computing the score of a data object.

Definition 7. *The **nearest neighbor preference score** $\tau_i(p)$ **of data object** p based on the feature set F_i is defined as: $\tau_i(p) = \{s(t) \mid t \in F_i : dist(p,t) \leq dist(p,t')\ \forall t' \in F_i$ and $sim(t, \mathcal{W}_i) > 0\}$*

The overall spatio-textual score $\tau(p)$ of data object p is defined as in the case of the range score, and the query returns the k objects with the highest score. Again, *STDS* treats nearest neighbor queries similarly as in Algorithm 2 with subtle changes. The range predicate is removed in line 5 and line 9, while the child entries are prioritized in the heap according to their minimum distance from the data object p.

Regarding *STPS*, Algorithm 3 is directly applicable for the nearest neighbor score by modifying $nextCombination(Q)$ of Algorithm 4 to return the best combination based on score $s()$, but without discarding combinations that have a *distance* $> 2r$, as also in the case of the influence score. The remaining challenge is given a combination $\mathcal{C}$ to retrieve the data objects that satisfy the nearest neighbor requirement.

Generally, it is more difficult compared to the other score variants to retrieve the data objects for a given combination $\mathcal{C}$. We need to retrieve all data objects for which the nearest neighbor t_i based on F_i belongs to $\mathcal{C}$. For each feature object t_i of $\mathcal{C}$, there exists a region in which all data points that fall into that region have t_i as their nearest neighbor. This region corresponds to the Voronoi cell [12] and this problem has been studied for finding reverse nearest neighbors [10]. Only the data objects in the intersection of all regions need to be retrieved. In fact, we compute incrementally the Voronoi cell for each feature object t_i of $\mathcal{C}$, which allows us to discard early combinations for which the intersection becomes empty. We omit further implementation details due to space limitations.

8. EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of our algorithms *STDS* and *STPS*, presented previously in Section 5 and Section 6 respectively, for processing spatio-textual preference queries over large disk-resident data. Moreover, we study the gains in performance of our algorithms caused by the SRT index proposed in Section 4 compared to an existing indexing technique (IR^2-Tree [8]). In order to ensure a fair comparison, we modify the IR^2-Tree to support score values of feature objects. To this end, we add to the leaf nodes of IR^2-Tree the scoring values for the feature objects, and maintain in ancestor (internal) nodes the maximum score of all enclosed feature objects. All experiments run on an Intel 2.2GHz processor equipped with 2GB RAM.

8.1 Experimental Setup

Methodology. In our experimental evaluation, we vary four important parameters of the datasets in order to study the scalability of the proposed techniques (Section 8.2). These parameters are: (i) the cardinality of the feature sets $|F_i|$, (ii) the cardinality of the set of data objects $|O|$, (iii) the number of feature sets c, and (iv) number of distinct keywords indexed. Moreover, we study four different query parameters to study how the characteristics of the query influence the performance of the algorithms (Section 8.3). In more details, we vary (i) the query radius r, (ii) the number k of retrieved data objects, (iii) the smoothing parameter λ between textual similarity and non-spatial score, and (iv) the number of keywords of the query for each feature set. Finally, we evaluate the performance of *STPS* for the influence score variant (Section 8.4) as well as for the nearest neighbor variant (Section 8.5).

Tested ranges for all parameters are shown in Table 2. The default values are denoted as bold. When we vary one parameter, all others are set to their default values.

Parameter	Range
Cardinality of dataset	$50K, \mathbf{100K}, 500K, 1M$
Cardinality of features sets	$50K, \mathbf{100K}, 500K, 1M$
Number of feature sets c	$\mathbf{2}, 3, 4, 5$
Indexed keywords	$64, \mathbf{128}, 192, 256$
Radius r (norm. in $[0,1]$)	$.005, \mathbf{.01}, .02, .04, .08$
k	$5, \mathbf{10}, 20, 40, 80$
Smoothing parameter	$.1, .3, \mathbf{.5}, .7, .9$
Queried keywords	$1, \mathbf{3}, 5, 7, 9$

Table 2: Experimental parameters.

Datasets. For evaluating our algorithms, we use both real and synthetic datasets. The real dataset, which was obtained from `factual.com`, describes hotels ($\approx$ 25K objects) and restaurants ($\approx$ 79K objects). In more details we collected restaurant and hotels that are annotated with their location. Moreover, for the collected restaurants we extracted their rating and their textual description of the served food, mentioned as "cuisine". The number of distinct values of keywords for the cuisine is around 130 and each restaurant description may contain one or more keywords. Our datasets contain collected hotels and restaurants for 13 US states that are the states for which `factual.com` lists sufficient data. In addition, we created synthetic clustered datasets of varying size, number of keywords and number of feature sets. Approximately $10,000$ clusters constitute each synthetic dataset. The number of distinct keywords is set to 256 as a default value and each feature object is characterized by one or more keywords that are picked randomly. The spatial constituent of all datasets has been normalized in $[0,1] \times [0,1]$. Every reported value is the average of $1,000$ random queries, which are generated in a similar way as the synthetic data and follow the same data distribution.

Metrics. The efficiency of all schemes is evaluated according to the average execution time required by a query (time consumed in the CPU and to read disk-pages). In our figures we break down the execution time into the time consumed due to the disk accesses (dark part of the bars) and the time needed for processing the query (CPU time) which is the white part of the bars. The time consumed due to the disk accesses relates to the number of the required I/Os.

8.2 Scalability Analysis

In this section, we evaluate the impact of varying different parameters on the efficiency of our algorithms. In order to perform a scalability analysis, we employ the synthetic dataset for this set of experiments. First, we show the scalability limitations of *STDS* for large datasets (Table 3), and then we explore in more detail the significantly superior performance of *STPS*.

Feature objects $\lvert F_i \rvert$	50000	100000	500000	1000000
IR^2-tree	13427.3	13854.6	25223.1	31434.6
SRT	12301.7	13187.9	18725.1	23046.3
Data objects $\lvert O \rvert$	50000	100000	500000	1000000
IR^2-tree	13073.2	13854.6	21074.2	27846.0
SRT	11718.1	13187.9	18267.4	23444.9
Number c of F_i	2	3	4	5
IR^2-tree	13854.6	27842.6	33625.0	40188.4
SRT	13187.9	14104.9	32071.1	38340.7
Indexed keywords	1	2	3	4
IR^2-tree	13698.7	13854.6	15655.6	16209.6
SRT	13121.4	13187.9	13207.9	13887.8

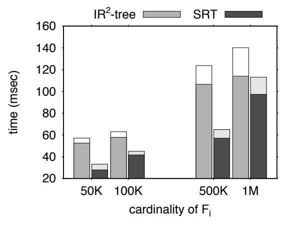
Table 3: *STDS* execution time (in msec) for synthetic dataset.

Table 3 shows the results for *STDS* when varying different parameters of the dataset. For the default setting, *STDS* requires over 13 seconds for range queries. Evidently, when a large number of data objects is involved *STDS* does not scale well and the absolute time required is high. The main reason is that *STDS* associates all data objects with c feature objects, which is particularly time-consuming. This experiment demonstrates that a plain algorithm for solving the problem can lead to prohibitive processing cost. Since *STDS* performs badly for all experimental setups, we omit *STDS* for the rest of experimental evaluation, and study the performance of *STPS* coupled with two different indexing techniques.

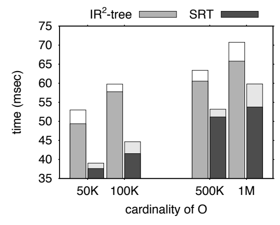
Figure 7 illustrates the results for the same experiment as above, but for the *STPS* algorithm. We implemented *STPS* over two different indexes: (i) our SRT index (proposed in Section 4), and (ii) the modified IR^2-Tree [8] whose nodes are enhanced with the maximum score of enclosed feature objects. In summary, the results clearly demonstrate that *STPS* scales with all parameters and that SRT indexing always outperforms IR^2-tree. Moreover, in both cases, the *STPS* algorithm exhibits high performance, as witnessed by the low execution time, which stems from its ability to quickly identify qualified feature combinations. Consequently, the significant gains in processing time (orders of magnitude compared to *STDS*) are mostly due to the effective design of the algorithm. The SRT index additionally offers a speedup of x2 compared to the IR^2-Tree, which further improves the overall performance.

Figure 7(a) shows the execution time when increasing the cardinality of the feature sets. *STPS* scales well since the execution time increases only by a factor of at most x3, when increasing the dataset by one order of magnitude. This increase is due to the increased size of the data structures and the additional processing required to traverse a bigger data structure and find valid combinations of high score. When comparing the index structures, the SRT index is faster, due to the clustering of all score constituents (distance, textual similarity, and non-spatial score) in the 4-dimensional space.

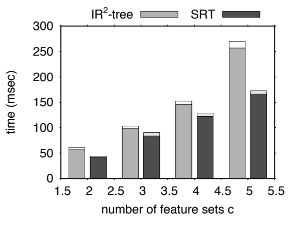
Figure 7(b) shows the obtained results when increasing the number of data objects. Again, *STPS* scales well, and, in fact, even better than in the previous experiment. Obviously, a larger dataset of data objects does not affect the performance so much as larger feature sets. Again, the use



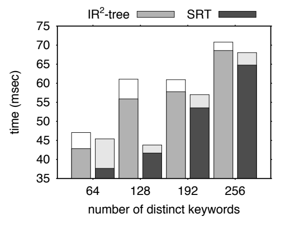
(a) Cardinality of F_i



(b) Cardinality of O

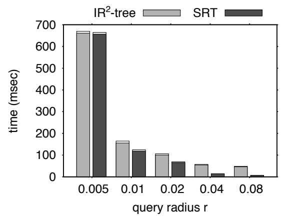


(c) Varying c

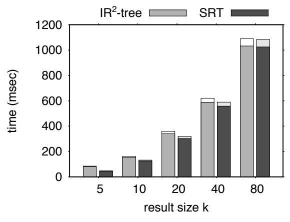


(d) Distinct keywords

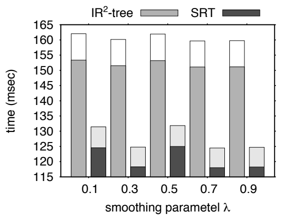
Figure 7: Scalability for synthetic dataset.



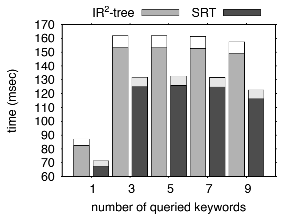
(a) Varying r



(b) Varying k



(c) Varying λ



(d) Queried keywords

Figure 8: Range query parameters for real dataset.

of SRT indexing consistently outperforms the IR^2-Tree.

In Figure 7(c), we increase the number of feature categories c. As expected, this has a stronger effect on performance, since the cost required to retrieve the highest ranked combinations increases with the number of possible combinations, which, in turn, increases exponentially with c. Still, the performance of *STPS* is not severely affected, especially in the case of the SRT index which scales gracefully with c.

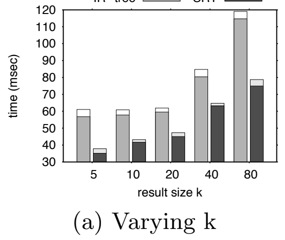
In Figure 7(d), we illustrate how the performance is affected by the number of distinct keywords in the dataset. Apparently, a higher number of keywords causes higher execution times. The reason is twofold. First, as the number of distinct keywords increases, it is less probable to find feature objects that are described by all queried keywords, thus more feature objects need to be retrieved in order to ensure that no other combination has a higher score. Secondly, the node capacity of the index structures drops, thus the height of the index structures may increase, thus causing more IOs. In any case, the increase in the absolute value of execution time is relatively small (20 msec), even when we increase the vocabulary by a factor of 4 (from 64 to 256 keywords).

8.3 Varying Query Parameters

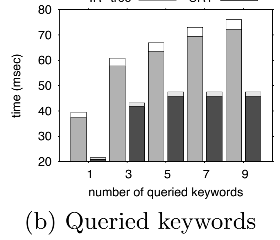
In Figure 8, we study the effect of varying query parameters for the real dataset. First, in Figure 8(a), we evaluate the impact of increasing the query radius r on the performance of *STPS*. We notice that for smaller values of r the execution time increases and the gain of SRT indexing compared to IR^2-tree drops. For small radius, access to more qualified combinations of feature objects is required, since only few data objects are located in their neighborhood. Therefore, for both indexing approaches the execution time increases mainly due to the increase of the IOs. Since a high percentage of the feature objects need to be retrieved for each feature set, the gain of SRT indexing is small. However, difference in performance becomes obvious for greater values of r, and hence, finding relevant feature objects in terms of textual description and good non-spatial score becomes most important for accessing only few feature objects.

Figure 8(b) illustrates the execution time when varying the size of result set k. Overall, the execution time increases as k increases. Specifically, with higher values of k more combinations of feature objects are retrieved to compose the result set, which again lead to more IOs to retrieve the qualifying feature objects that constitute valid combinations.

In Figure 8(c), we vary the smoothing parameter λ. In general, both approaches exhibit relatively stable performance for varying values of λ. The performance of IR^2-tree is not affected by the smoothing parameter, since the feature objects are not grouped into blocks based on the non-spatial score nor based on their textual similarity. We note for the IR^2-tree that objects with similar textual descriptions are stored throughout the index, regardless of their non-spatial score; unlike the SRT index where they are clustered together in the same block. As a result, a significant overhead is evident when searching for relevant objects all over the IR^2-tree. On the other hand, the SRT index is built by taking into account non-spatial score, the textual information and the spatial location. Thus, *STPS* that uses SRT index is consistently more efficient regardless of the value of the smoothing parameter.



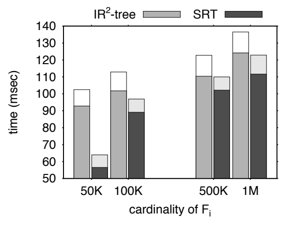
(a) Varying k



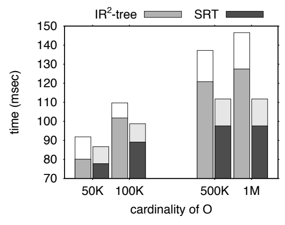
(b) Queried keywords

Figure 9: Query parameters for synthetic dataset.

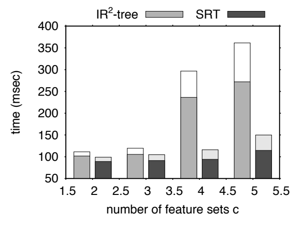
In Figure 8(d), we vary the number of queried keywords per feature set from 1 to 9. The number of queried keywords



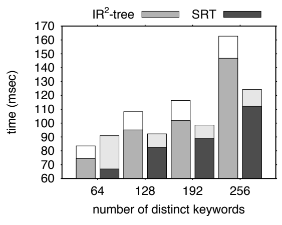
(a) Cardinality of F_i



(b) Cardinality of O

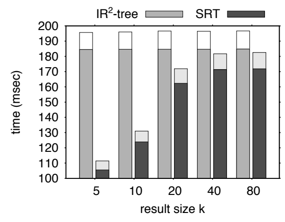


(c) Varying c

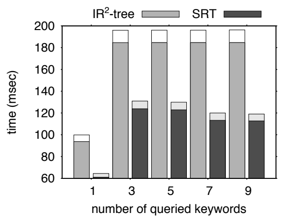


(d) Distinct keywords

Figure 10: Scalability for synthetic dataset and infuence queries.

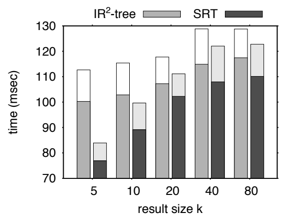


(a) Varying k

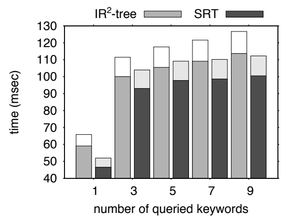


(b) Queried keywords

Figure 11: Influence query for real dataset.



(a) Varying k



(b) Queried keywords

Figure 12: Influence query for synthetic dataset.

has little impact on performance, except for the special case where one keyword is queried for each feature set. This is because both of the indexing techniques aggregate in the non-leaf nodes the textual information of the leaf nodes, which makes it much easier to find objects that contain one keyword, rather than finding objects that are described with more keywords. Nevertheless, the gain in execution time of SRT indexing compared to the IR^2-tree is obvious.

Figure 9 depicts results obtained from the synthetic dataset, when varying different query parameters. We notice the same tendency as in the case of the real dataset. In general, we observed that range queries are costlier for the real dataset. This is due to the data distribution: our real dataset, which was extracted from `factual.com`, consists of restaurants and hotels in the US forming just a few clusters. On the other hand, our synthetic dataset is substantially larger and contains a few thousands of clusters. Hence, the data from the latter dataset are more dispersed compared to the former. Last but not least, the SRT indexing consistently outperforms the IR^2-tree.

8.4 Influence-based Preference Score

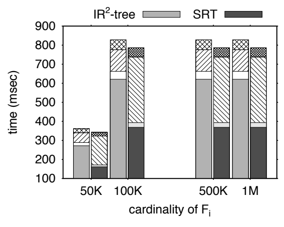
In this section, we study the performance of *STPS* for the influence-based score variant of the spatio-textual preference queries. Figure 10 shows the scalability analysis of *STPS* for this query variant. By comparing the results to Figure 7, which studies the execution time of the range score variant for the same parameters, we conclude that the required execution time is comparable and in some cases slightly increased. This is because more data object for each combination must be retrieved (for the influence-based score variant), since data objects that are further away than r may also have a non-zero score. Nevertheless, the additional cost is not significant in our experiments, and we notice the same tendency in execution time as in the case of range score, thus similar conclusions can be drawn. Moreover, the SRT indexing technique is beneficial in all setups.

Figure 11 shows the execution time of *STPS* for the real dataset when varying query parameters. In Figure 11(a), time decreases for large k values compared to the range score (Figure 8(b)), because combinations with high score are associated with all data objects. Even though the score of the object is reduced based on the distance, still their score is high enough to retrieve fewer combinations. For smaller k values the execution time is not affected significantly. In Figure 11(b), we evaluate the performance of *STPS* when varying the number of queried keywords. We notice that the execution time is similar to Figure 8(d), which depicts the results of the same experiment for range score.

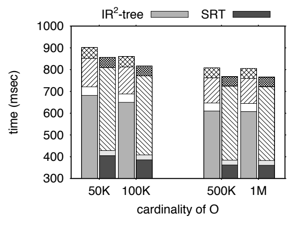
Finally, in Figure 12, we study the performance of *STPS* for the synthetic dataset when varying query parameters. The execution time is similar and slightly higher to the execution time needed for the range score (Figure 9), while the behavior of *STPS* when varying query parameters is the same. Again, the SRT indexing technique improves the performance of *STPS* consistently.

8.5 Nearest Neighbor Preference Score

In this section, we evaluate the performance of *STPS* for the nearest neighbor score variant. In general, we noticed that the execution time is higher compared to the other score variants, which is due to the Voronoi cell computations required for retrieving the data objects. In the charts, we illustrate separately with a striped pattern the IO (lower striped part) and the CPU-time (upper striped part) required to compute the respective Voronoi cells. Moreover, it is expected that for a given combination, few data objects satisfy the nearest neighbor constraint, which leads to retrieval of more combinations compared to the other variants. Therefore, we notice in the charts that the execution time is high even if the Voronoi cell computations is not considered (without stripped parts). We note that for static data the Voronoi cells can be pre-computed in a special structure, and therefore significantly reduce the execution time.

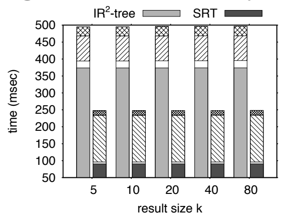


(a) Cardinality of F_i

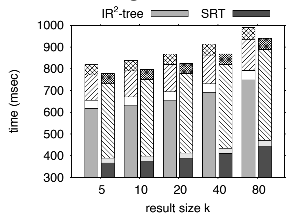


(b) Cardinality of O

Figure 13: Scalability of nearest neighbor variant.



(a) Real dataset



(b) Synthetic dataset

Figure 14: Varying k for nearest neighbor variant.

Figure 13 depicts the execution time for *STPS* for the synthetic dataset, while varying the size of the feature and object datasets. In Figure 13(a) we notice that for large feature sets the dominant cost is finding the data objects for a given combination (i.e., computing the Voronoi cells), rather than retrieving the combination with the highest score. Computing the Voronoi cells requires retrieval of feature objects from the spatio-textual index of F_i to define the borders of the cell. This cost is higher for the SRT indexing method compared to the IR^2-tree, since the IR^2-tree is built based on spatial information only and nearby feature objects are stored in the same node. Nevertheless, SRT indexing is still beneficial for *STPS*, but the gain is smaller than for the other variants. Similar conclusions can be drawn when varying the cardinality of the dataset O, as depicted in Figure 13(b).

In Figure 14, we vary the parameter k both for real (Figure 14(a)) and synthetic datasets (Figure 14(b)). We notice that the execution time does not increase significantly when increasing k for the real dataset. This is because there exist some combinations for which their feature objects are the nearest neighbor for many data objects. Thus, the same effort is needed for retrieving few or many data objects. This is not the case for the synthetic dataset (Figure 14(b)), where the execution time increases for higher values of k.

9. CONCLUSIONS

Recently, the database research community has lavished attention on spatio-textual queries that retrieve the objects with the highest spatio-textual similarity to a given query. Differently, in this paper, we address the problem of ranking data objects based on the facilities (feature objects) that are located in their vicinity. A spatio-textual preference score is defined for each feature object that takes into account a non-spatial score and the textual similarity to user-specified keywords, while the score of a data object is defined based on the scores of feature objects located in its neighborhood. Towards this end, we proposed a novel query type called *top-k spatio-textual preference query* and present two query processing algorithms. *Spatio-Textual Data Scan* (*STDS*) first retrieves a data object and then computes its score, whereas *Spatio-Textual Preference Search* (*STPS*) first retrieves highly ranked feature objects and then searches for data objects nearby those feature objects. Moreover, we proposed an indexing technique that improves the performance of our algorithms. Furthermore, we show how our algorithms can support different score variants. Finally, in our experimental evaluation, we put all methods under scrutiny to verify the efficiency and the scalability of our method for processing top-k spatio-textual preference queries.

Acknowledgments

The work of A. Vlachou was supported by the Action "Supporting Postdoctoral Researchers" of the Operational Program "Education and Lifelong Learning" (Action's Beneficiary: General Secretariat for Research and Technology), and is co-financed by the European Social Fund (ESF) and the Greek State.

10. REFERENCES

[1] P. Bouros, S. Ge, and N. Mamoulis. Spatio-textual similarity joins. *PVLDB*, 6(1):1–12, 2012.

[2] X. Cao, G. Cong, and C. S. Jensen. Retrieving top-k prestige-based relevant spatial web objects. *PVLDB*, 3(1):373–384, 2010.

[3] X. Cao, G. Cong, C. S. Jensen, and B. C. Ooi. Collective spatial keyword querying. In *Proc. of SIGMOD*, pages 373–384, 2011.

[4] X. Cao, G. Cong, C. S. Jensen, and M. L. Yiu. Retrieving regions of interest for user exploration. *PVLDB*, 7(9):733–744, 2014.

[5] L. Chen, G. Cong, C. S. Jensen, and D. Wu. Spatial keyword query processing: An experimental evaluation. *PVLDB*, 6(3):217–228, 2013.

[6] G. Cong, C. S. Jensen, and D. Wu. Efficient retrieval of the top-k most relevant spatial web objects. *PVLDB*, 2(1):337–348, 2009.

[7] Y. Du, D. Zhang, and T. Xia. The optimal location query. In *Proc. of SSTD*, pages 163–180, 2005.

[8] I. D. Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In *Proc. of ICDE*, pages 656–665, 2008.

[9] I. Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In *Proc. of VLDB*, pages 500–509, 1994.

[10] F. Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. In *Proc. of SIGMOD*, pages 201–212, 2000.

[11] Z. Li, K. C. K. Lee, B. Zheng, W.-C. Lee, D. L. Lee, and X. Wang. IR-tree: An efficient index for geographic document search. *IEEE TKDE*, 23(4):585–599, 2011.

[12] A. Okabe, B. Boots, K. Sugihara, and S. N. Chiu. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams.* John Wiley and Sons Ltd., New York, NY, 2000.

[13] J. B. Rocha-Junior, O. Gkorgkas, S. Jonassen, and K. Nørvåg. Efficient processing of top-k spatial keyword queries. In *Proc. of SSTD*, pages 205–222, 2011.

[14] J. B. Rocha-Junior, A. Vlachou, C. Doulkeridis, and K. Nørvåg. Efficient processing of top-k spatial preference queries. *PVLDB*, 4(2):93–104, 2010.

[15] T. Xia, D. Zhang, E. Kanoulas, and Y. Du. On computing top-t most influential spatial sites. In *Proc. of VLDB*, pages 946–957, 2005.

[16] M. L. Yiu, X. Dai, N. Mamoulis, and M. Vaitis. Top-k spatial preference queries. In *Proc. of ICDE*, pages 1076–1085, 2007.

[17] M. L. Yiu, H. Lu, N. Mamoulis, and M. Vaitis. Ranking spatial data by quality preferences. *IEEE TKDE*, 23(3):433–446, 2011.

[18] D. Zhang, Y. M. Chee, A. Mondal, A. K. H. Tung, and M. Kitsuregawa. Keyword search in spatial databases: Towards searching by document. In *Proc. of ICDE*, pages 688–699, 2009.

Probabilistic Resource Route Queries with Reappearance

Gregor Jossé, Klaus Arthur Schmid, Matthias Schubert
Institute for Informatics, Ludwig-Maximilians-University Munich
Oettingenstr. 67, 80538 Munich, Germany
{josse,schmid,schubert}@dbs.ifi.lmu.de

ABSTRACT

In many routing applications, it is unclear whether driving to a certain destination yields the desired success. For example, consider driving to an appointment and looking for a parking spot. If there are generally few parking spots in the area or if occupancy of spots is currently high, the search may not be successful. In this case, the search is continued, possibly into a different area, where chances of success are higher. We generalize this problem and introduce a probabilistic formalization to model the availability of resources at certain locations. Our probabilistic model considers short term observations (e.g., vacant parking spots) as well as long term observations (e.g., average occupancy time) to adapt to the level of information currently available. In contrast to previous models, we allow resources to reappear after a probabilistically modeled amount of time (e.g., a car leaves a spot). Based on this model, we propose the so-called probabilistic resource route query with reappearance. In order to compute feasible solutions to this query in interactive time, we propose two greedy approaches. Furthermore, we examine backtracking for computing exact solutions and extend the proposed method into a significantly more efficient branch and bound algorithm. In our experiments, we investigate two realistic applications, examine the benefit of our model, and compare algorithmic solutions w.r.t. result quality and computational efficiency.

1. INTRODUCTION

With increasing gas prices, escalating greenhouse gas emissions and heavy traffic congestion in metropolitan areas, optimizing traffic is of great ecological and social importance. While the basic routing task of finding a path from start to target is a well-explored research area, there are other routing tasks common in everyday life which have drawn less attention so far. An example are trip planning queries (TPQ) which are specified by a start location, a target location and a number of resource types which have to be visited along the trip. For instance, the user might provide the resource types "ATM", "gas station", and "department store". The result of such a query is the shortest path from start to target visiting exactly one instance of each resource type. There are several variants to this kind of problem which will be reviewed in Section 2.

A problem relevant to everyday life is guiding a user through a road network to enable them to find a resource for which the availability at certain locations is uncertain. There are several examples for this task. For instance, consider parking spots: although it might be known that certain streets allow parking, it is generally not known whether there will be any vacant spots upon arrival. Another example are drivers of electric cars looking for public charging stations. In some developing countries, hospitals with emergency capacities are scarce. Thus, ambulances often visit more than one hospital before being able to hospitalize their patient. In all of these cases, it holds that if the resource is not available upon arrival, the search must be continued to other resource locations until an available resource is found. Hence, guiding a user to the closest resource does not yield a satisfactory solution. Instead, in order to yield a sufficiently large probability of success, a route visiting several resource locations is required. A general problem of finding such routes is that there are two contrary characteristics describing the quality of a route. The first quality measure is the overall likelihood of finding an available resource when following the route. The second quality measure is the cost, e.g., the expected travel time or distance, until the respective resource is found. These two measures are complementary, because continuing the search to another resource location will always increase the success probability but also – without exception – the cost. Thus, it is not possible to minimize the cost while maximizing the probability of success.

In order to compute the success probability of a route, it is necessary to employ information about the availability of resources at all known resource locations. In this paper, we assume a system which collects observations on the availability (and conversely the consumption) of resources over time. These so-called long term observations are then used to compute probability distributions modeling general resource availability. Furthermore, the system provides current information about resource status at query time. We refer to this kind of information as short term observations. For example, long term observations correspond to the average time a parking spot remains vacant or occupied. Short term observations, on the other hand, provide information about currently vacant spots. Depending on the scenario, the amount of short term observations might be limited, e.g., only a limited number of parking spots are equipped with occupancy sensors while the rest is detected and reported by other roaming cars. At first glance, short term observations might seem sufficient for a successful search. However, knowing that a resource is available at the moment, does not mean that it still will be upon arrival. Thus, as time progresses, the influence of short term observations on the probability of finding a resource decays. Long term observations address this problem in three ways. First, if there is no short term observation available for a resource, the expected vacancy time of a resource can compen-

10.5441/002/edbt.2015.39

sate for the lack of current information. Furthermore, long term observations can be used to predict the probability that a currently available resource will still be vacant upon arrival. And, finally, long term observations can serve as an estimate for the expected occupancy time, i.e., the expected time until a consumed resource becomes available again. For example, in the parking spot scenario this corresponds to the expected parking duration.

In this paper, we present a statistical model describing a road network comprising resource locations of a specific resource type. Our model incorporates both types of information described above, i.e., long term as well as short term observations. From a formal point of view, our model describes each resource location as a continuous-time Markov chain with two states, `available` and `consumed`. Based on this model, we introduce the following query: For a given query location, compute a route for which the probability of finding an available resource exceeds a given probability threshold (e.g., 90 %) while minimizing the (expected) cost, e.g., travel time or distance. A route may be extended infinitely, and each extension adds to the success probability but also to its cost. Thus, in order to optimize one measure, we have to bound the other. More precisely, the best route may be the one with the least cost among all routes exceeding the probability threshold. Or, converesely, the best route may be the one with the highest success probability among all routes not exceeding a cost threshold.

Since our model allows for reapparance of resources, the search space of possible solutions is unlimited. However, in many applications the time frame of finding a suitable answer is rather small as users only tolerate limited answering time. Therefore, we investigate two greedy search heuristics which promise admissible results in efficient time. To allow the computation of optimal results, we examine a recursive backtracking approach to avoid exhaustive search. Furthermore, we propose a lower bound for the remaining increase in cost used in a highly efficient branch and bound solution providing optimal results. We evaluate our approach within real world road networks on the application of finding parking spots and on the application of finding charging stations for electric vehicles. To conclude, the contributions of this paper are as follows:

- A novel probabilistc model describing the availability of resources in road networks. We model resources as continuous-time Markov chains which are parametrized by long term as well as short term observations and allow to model the reapparance of previously consumed resources.
- A new type of query, the Probabilistic Resource Route Query with Reappearance (PRRQR).
- An approximate solution to the PRRQR employing two different search heuristics, as well as optimal solutions based on backtracking and branch and bound.

The rest of the paper is organized as follows: Section 2 summarizes related work about similar types of queries. In Section 3, our probabilistic model is described, followed by the formal definition of the PRRQR. Section 5 describes the heuristics, bounds and query algorithms introduced to process PRRQRs. The results of our experimental evaluation are presented in Section 6. The paper is concluded with a summary and an outlook for future work in Section 7.

2. RELATED WORK

In this section, we survey existing work on similar tasks. First, we will give a review of basic query types related to the one introduced in this paper. Then, we address works which model the existence of resources in a probabilistic way. All of the following query types have the same meta-task, namely, guiding a user to a certain resource. In all of these scenarios, a database which stores the resources and their respective locations is assumed. Although this task may also be carried out in Euclidean space, we restrict ourselves to road networks, as most of the applications are traffic-related.

The simplest type of query guiding a user to the next available resource is the nearest neighbor query (NNQ). In this setting, the user specifies a location – typically his current location – as well as some type of resource. The result of the NNQ is the optimal path (fastest, shortest, etc.) to the closest location providing the resource. As NNQ are a well-explored research area, we will not go into detail on their solutions. An extension of this query are trip planning queries (TPQ) [2] (also referred to as route planning queries [1] or route search queries [10]). In this problem setting, the user specifies a selection of different resources, e.g., “ATM”, “restaurant”, “florist”, “cinema”. Additionally to his start location the user may specify a target location. The result of a TPQ is the optimal path from start to target visiting at least one instance of each resource. Computing TPQs is NP-complete because in the case that each specified resource type occurs exactly once the TPQ degenerates to the Traveling Salesman Problem (TSP).

In another variation of the problem, the order in which resources are visited may be constrained, as described in [9] or [7]. For instance, if planning a date, the order of resources might be restricted by the constraints that the ATM has to be visited first and the florist should be visited before the restaurant and the cinema. However, since the task usually maintains an NP-hard subproblem, solutions to any of these problems typically employ heuristics ([1], [9]).

In the settings presented so far, the existence of a resource at its location is considered to be guaranteed. In many real world applications, however, a resource will only be available with a certain probability. If no table or seats have been reserved, not all locations of type “restaurant” or “cinema” might have the resource available. The same holds for the resource type “florist” if looking for specific flowers. In all of these cases, the requested resource is available with a certain probability (and consumed with the converse probability), i.e., prior to arrival it is not known with certainty whether the resource is available or consumed. At first glance, these kinds of uncertainty may seem congruent. However, there are significant differences which require specific modeling. We distinguish the following types of uncertainty.

Assume a surfer is looking for good waves and is considering different beaches. At every beach, the waves might be sufficient with a certain probability. If available, the resource “waves” cannot be consumed by the presence of other surfers. Thus, we refer to the probability of finding waves as *static (resource) uncertainty*. This uncertainty is independent from the time of arrival and the presence of competitors.

Now, consider a cinema where seats are a limited resource. If no seats have been reserved, the probability of finding seats for a particular show decays as screening time approaches. Since in this case a limited quantity of the resource is consumed over time, we refer to this type of uncertainty as *time-decaying (resource) uncertainty*. Contrastingly, while all tables at a certain restaurant might be occupied now, there might be one available later the same evening. In this case, the quantity of the resource might decay (or more generally: change) over time but regenerate at a later point in time. This is a significant difference to the other scenarios where revisiting resources does not yield any benefit. However, in this scenario, although the resource might have been consumed upon arrival, it might make sense to revisit after an adequate waiting time.

We refer to this kind of uncertainty as *time-dependent (resource) uncertainty with reappearance*.

To the best of our knowledge, there is no previous work supporting short term observations or taking time-dependent uncertainty with reappearance into account. Therefore, we shall now review works which incorporate static as well as time-decaying resource uncertainty. While we provide an abstract problem definition (cf. Section 3) and applications based upon this definition, some works focus on their application and adapt their problem definition to the respective use case. Nevertheless, these works can – with some restrictions – in many cases be extended to incorporate general resources.

The authors of [10], for example, compute paths which guide the user along certain resources. In their follow-up work, [7], this problem is extended by ordering constraints (as in some of the examples above). Both papers present algorithms based on greedy search heuristics as well as on heuristics which minimize the expected distance until search success. In both papers, resources may have assigned success probabilities. However, these probabilities reflect static uncertainty, i.e., non-time-dependent and non-reappearing. The same holds for [4] and its follow-up work [8] where probabilistic k-route queries are introduced and examined. Here, the authors introduce a confidence value which corresponds to the existence probability of a resource at each location, also reflecting static uncertainty. Employing different heuristics, the presented algorithms find approximate solutions by clustering resource locations that maximize the expected success.

In contrast, the authors of [3] take time-dependency into account and assume a linear decay for the vacancy of parking spots. Although this model is aimed at a specific application, it may be generalized to abstract resources. Note, however, that this model does not allow reappearance of resources which is a significant shortcoming, especially in this application. This is because a typical strategy looking for a parking spot is roaming the target area until someone vacates a spot, i.e., a resource reappears. The authors propose two approaches to maximizing the probability of finding a parking spot. The first approach finds the optimal result, however, this is done employing full enumeration on the time-varying TSP. Due to the brute force nature of this algorithm, query processing quickly becomes infeasible with increasing number of resource locations. The second approach is an algorithm that clusters resource locations before solving a TSP on the clusters. Subsequently, the optimal solution within each cluster is searched. Based on a heuristic, this algorithm yields an approximation of the optimal result, while providing a considerable speed-up.

There are various methods, focusing on the application of taxi pickups as well as ridesharing, such as [12], [14],[11] and [5]. In [12], the task is equivalent to solving a classic TSP, i.e., ordering different but fixed pickup locations such that the total distance is minimized. The authors rely on a genetic algorithm approach to solve this problem. In [14],[5] and [11] the task is more complicated. Here, the task is first of all to assign cabs to a set of currently available customers. After this assignment is done, a route for picking up and dropping off the customers has to be found. Thus, only the last part of the query is related to our work. Furthermore, none of the works considers uncertainty w.r.t. customer availability.

3. PROBLEM SETTING

In this section, we formalize our problem setting. First, we define the graph which represents the underlying road network. Then, we introduce the probabilistic model which describes resource availability and consumption.

3.1 Road Network Graph

For a given road network, we let $G = (V, E)$ denote the corresponding graph, i.e., the vertices (or nodes) $v \in V$ correspond to crossings, dead ends, etc., and the edges $e \in E \subseteq V \times V$ represent directed road segments connecting the vertices. We refer to this graph as *Road Network Graph*. Furthermore, let $c : E \to \mathbb{R}_0^+$ denote the function which maps every edge onto its respective cost, e.g., travel time or distance. If the employed cost function is not travel time, we additionally assume the travel time to be known and given by a function $t : E \to \mathbb{R}_0^+$ (as resource availability is dependent on the time of arrival). By *route*, we mean a consecutive set of edges (possibly with cycles), i.e., $r = (e_1, \ldots, e_n)$ where all e_i are taken from the corresponding set of edges and for all $1 \leq j \leq n-1 : e_i = (u, v) \Rightarrow e_{i+1} = (v, w)$. The cost of a route $r = (e_1, \ldots, e_n)$ is defined as the accumulated cost of its edges, i.e., $c(r) = \sum_{i=1}^{n} c(e_i)$. By *path*, we mean a cycle-free route.

3.2 Probabilistic Model

In the following, we introduce our probabilistic model. As mentioned before, at every resource location the respective resource may either be `available` or `consumed`. However, prior to arrival at the location, it is not known which of the two is the case. Our probabilistic model has to be able to reflect all three kinds of uncertainty: static, time-decaying, and time-dependent uncertainty with reappearance. While the former two kinds of uncertainty are rather straightforward, the latter requires more work and a novel approach.

The static uncertainty of a resource is easily expressed by a random variable X which takes the values 0 or 1, representing the states `available` and `consumed`, respectively, where the probabilities of X are $\mathbb{P}(X = 0) = p$ and $\mathbb{P}(X = 1) = 1 - p$ for some $p \in [0, 1]$. For illustration, recall the example of a surfer looking for waves at a beach. Independent of the time of their arrival there will be waves with probability p.

In the case of time-decaying uncertainty, we propose modeling the probability that a resource X is available at time $t > 0$ as $e^{-\lambda t}$ for some $\lambda > 0$. Consequently, at time $t = 0$, the resource is available with probability 1, but it decreases as time progresses and asymptotically approaches 0. Or, in other words, the probability that X is consumed is the cumulative distribution function of an λ-exponentially distributed random variable. This coincides with the intuition of modeling waiting times as exponentially distributed random processes. For illustration, recall the example of buying tickets to the movies, where the probability of available seats is 1 at $t = 0$ but decreases as screening time approaches.

Now, let us turn to the case of time-dependent uncertainty with reappearance which is the core of this work, as it is the only concept that can model the use cases of our experiments (parking spots, charging stations). As before, resource availability has two states, but now, there may occur multiple state transitions at arbitrary points in time. Therefore, we propose modeling each resource as a stochastic process. The most common type of stochastic processes are Markov chains which model the transition probabilities within a system with a given number of states. When a system transitions from one state into another, the future state is only dependent on the present state. This property is central to Markov chains and referred to as memorylessness or Markov property. Markov chains can either assume discrete or – as in our case – continuous time. In a discrete model, there exist equal time steps, and for each step, the probability of transitioning into another state can be computed. In a continuous-time model, the sojourn time in each state, i.e., the time until the next state transition, is perceived as a random variable itself. The notion of memorylessness extends naturally to the case of continuous time.

Thus, we model time-dependent resource availability with reappearance at each resource location as a continuous-time Markov chain (CTMC). More precisely, each resource location r^i is now represented by a family of random variables $\{X_t^i, t \geq 0\}$ with values in the state set $\{0, 1\}$. Note that there exists a one-to-one relationship between each resource location, and its resource modeled by the respective CTMC. Thus, we denote the CTMC of each resource location r^i by X^i and denote the set of all CTMCs by $\mathcal{X}$, where $|\mathcal{X}| = |\mathcal{R}|$. We may use the term *resource* for the geolocation associated with a resource as well as for the corresponding CTMC. When not clear from the context, we will state explicitly which of the two is referred to. Also, we assume the resources, more specifically their CTMCs, to be mutually independent. The independence assumption keeps the model general and applicable even if the available observations are limited.

Besides its state space, a CTMC is (under the reasonable assumption of time-homogeneity) defined by the family of transition matrices $\{P_t, t \geq 0\}$ and the (infinitesimal) generator matrix Q. Using the Kolmogorow equations, each may be computed from the other by solving the first order differential equation $P'(t) = P(t)Q$. For more mathematical details, we refer the reader to [13]. We omit the explicit calculations here and restrict ourselves to explaining the connection between $P(t), Q$ and the states of a resource.

In our case, Q is a 2×2-matrix. Its diagonal entries reflect the parameters of the random variables modeling the sojourn time of each state while the non-diagonal entries reflect the rate of transition into another state. Q has the following form:

$$Q = \begin{pmatrix} -\lambda & \lambda \\ \mu & -\mu \end{pmatrix}$$

The family of transition matrices $P(t)$ for $t \geq 0$ is defined as follows:

$$P(t) = \begin{pmatrix} \frac{\mu}{\lambda+\mu} + \frac{\lambda}{\lambda+\mu} e^{-(\lambda+\mu)t} & \frac{\lambda}{\lambda+\mu} - \frac{\lambda}{\lambda+\mu} e^{-(\lambda+\mu)t} \\ \frac{\mu}{\lambda+\mu} - \frac{\mu}{\lambda+\mu} e^{-(\lambda+\mu)t} & \frac{\lambda}{\lambda+\mu} + \frac{\mu}{\lambda+\mu} e^{-(\lambda+\mu)t} \end{pmatrix} \quad (1)$$

For each resource, the sojourn times of its states `available` and `consumed` are modeled as exponentially distributed random variables with parameters λ and μ, respectively. This, again, coincides with the convention of modeling waiting times as exponentially distributed. The expected value of an exponential distributed random variable $\exp(\phi)$ is $1/\phi$. Thus, the expected sojourn time in the state of availability is $1/\lambda$, and the expected sojourn time in the state of consumption is $1/\mu$. One application on which we evaluate our model in Section 6 is finding vacant parking spots. In this use case, $1/\lambda$ would be the expected vacancy time, analoguously, $1/\mu$ would be the time until an occupied spot becomes vacant again. Of course, these parameters may be different for each resource location if enough observations for an individual estimation are available. Therefore, it is possible for each resource to have distinctly parametrized Q and $P(t)$.

Let us shortly explain how the assumed observations are used for parameter estimation. As mentioned before, our model allows to incorporate short term as well as long term observations. The latter are used for parameter estimation in the following way: Consider a resource X, which has an unknown expected sojourn time in the state `available` but is assumed to be exponentially distributed with parameter λ. Given a number of observations $x = (x_1, \ldots, x_r)$, i.e., exemplary measurements of the time span during which X stays available, we can easily estimate λ using the maximum likelihood estimator. The according likelihood function is given by:

$$L(\lambda) = \prod_{i=1}^{r} \lambda \exp(\lambda x_i) = \lambda^r \exp(-\lambda r \bar{x})$$

where $\bar{x} = 1/r \sum x_i$ denotes the mean of all measurements. Differentiating the logarithmized likelihood function yields the maximum likelihood estimator $\hat{\lambda} = 1/\bar{x}$, which is simply the inverse of the mean value. The parameter μ may of course be estimated analoguously.

Now, let us review some properties of the transition matrices $\{P(t), t \geq 0\}$ central to the model. Given a resource X and its sojourn time parameters λ and μ, we can compute $P(t)$ as in Equation 1. If we also have an initial probability distribution based on short term observations of the states of X at time $t_0 = 0$ (denoted as π_0), we can compute the according probability distribution after an arbitrary point in time $t \geq 0$ (denoted as π_t) as follows:

$$\pi_t = \pi_0 P(t)$$

Note that $P(0) = I$ is the identity matrix (cf. Equation 1) which means that if no time has passed, the probability distribution of t_0 is still active. For example, if there is an observation at t_0 of X being in state `consumed`, then $\pi_0 = (0, 1)$. The consumption of resource X is certain – but only at this particular point in time. As time progresses, it becomes more likely that the state changes. Therefore, the original probability distribution π_0 (given by the observation) changes. Note that this reflects the notion of reappearance of previously consumed resources. This is expressed in the (exponentially) decaying influence of the second summands in every entry of $P(t)$ (cf. Equation 1). Eventually, the original observation becomes obsolete. This can be seen from the convergence of $P(t)$ as $t \to \infty$:

$$\lim_{t\to\infty} P(t) = \begin{pmatrix} \frac{\mu}{\lambda+\mu} & \frac{\lambda}{\lambda+\mu} \\ \frac{\mu}{\lambda+\mu} & \frac{\lambda}{\lambda+\mu} \end{pmatrix}$$

Asymptotically both rows of $P(t)$ are equal. This implies the initial observation has no more influence on the probability distribution of X as $t \to \infty$. For example, whether the initial observation was `consumed`, i.e., $\pi_0 = (0, 1)$, or `available`, i.e., $\pi_0 = (1, 0)$ is without significance. In any case, $\lim_{t\to\infty} \pi_t$ is the so-called stationary distribution as introduced below.

In our case, a resource X is a finite-state Markov chain where all states communicate and thus X has a unique stationary distribution π [13]. By definition: $\pi P(t) = \pi, \forall t \geq 0$. Solving this system of equations, we get:

$$\pi = (\pi_1, \pi_2) = \begin{pmatrix} \frac{\mu}{\lambda+\mu} & \frac{\lambda}{\lambda+\mu} \end{pmatrix}$$

This is equal to the rows of $\lim_{t\to\infty} P(t)$ which supports the intuition of t having no more influence on the probability distribution. For example, if no observation for X is available, the only unbiased assumption is the stationary distribution since it assumes the respective share of both states w.r.t. λ and μ.

Let us use all of the above in an example related to one of our applications: Let X be a CTMC modeling the vacancy of a parking spot at a certain location. We make the following assumptions on the model: If `available` (meaning vacant), we expect X to be `consumed` (meaning occupied) within 5 minutes. This means, the sojourn time of state `available` is a $1/5$-exponentially distributed random variable. If X is `consumed`, we expect the occupant to leave within 20 minutes. Hence, the sojourn time of state `consumed` is a $1/20$-exponentially distributed random variable. As mentioned before, $P(0) = I$. Let us investigate how $P(t)$ changes as time (non-infinitely) progresses. For instance, after 1 and after 3.5 minutes:

$$P(1) \approx \begin{pmatrix} 0.8 & 0.2 \\ 0.05 & 0.95 \end{pmatrix} \quad P(3.5) \approx \begin{pmatrix} 0.52 & 0.48 \\ 0.12 & 0.88 \end{pmatrix}$$

Assume that at $t_0 = 0$ X has been observed as `available`, i.e., $\mathbb{P}(X_0) = (1, 0)$. Then at $t = 1$ the spot will still be `available` with probability 0.8. After 3.5 minutes this probability will have decreased to 0.52. Now, consider a different scenario where at $t = 0$ X has been observed as `consumed`, then after 1 minute it is `available` with probability 0.05. After 3.5 minutes this probability will have increased to 0.12.

To conclude, we now have a probabilistic model on our hands which is capable of describing all three kinds of resource uncertainty introduced in Section 2. The core contribution of this model is its ability to reflect resource reappearance. Also, it allows efficient resource-specific parametrization by incorporating long term and short term observations.

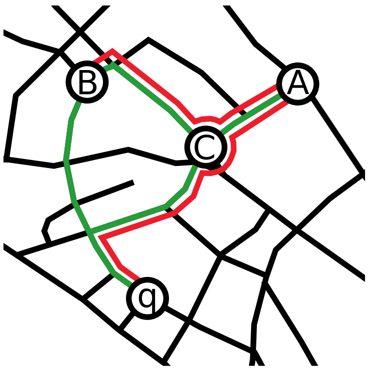
4. QUERY DEFINITION AND RESULT SET

Now that we have defined the probabilistic model, we turn to our query and its result. Both are best described using an alternative graph, referred to as *resource graph* $\hat{G}$. Therefore, in this section, we will first define the resource graph. Subsequently, we introduce two measures which are then used to define the Probabilistic Resource Route Query with Reappearance (PRRQR) and its result.

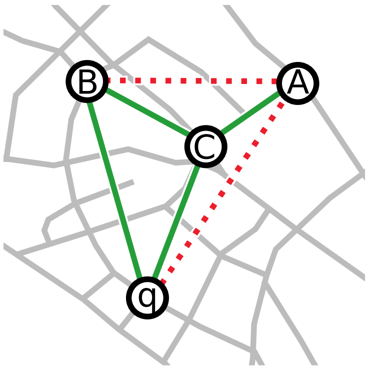
4.1 Resource Graph

We assume that for a road network graph and a query node q a set of suitable resources $\mathcal{X}(q) = \{X^1, \dots, X^N\}$ is given. In the majority of applications, only a reasonable subset of resources might qualify. For example, parking spots should be within walking distance of the driver's destination. In this case, the query node would be the driver's position when he reaches the vicinity of his destination. An according range query to a database would then retrieve suitable parking opportunities on which a PRRQR would be executed. For every resource X^i, we assume distribution parameters λ^i, μ^i and possibly an initial distribution π_0^i as given.

The set of vertices of the resource graph is defined as $\hat{V} := \{q\} \cup \mathcal{X}$, where $q \in V$ denotes the query node. The edges of the resource graph, $\hat{E}$, represent cost-optimal paths between resource locations in the underlying road network. Thus, each route in the resource graph can be expanded into a corresponding route in the transportation network (cf. Figure 1(a)). Let us note that even in cases where cost does not refer to the travel time, we will also compute the travel time of every cost-optimal path because it is needed to determine the success probability. Although it is possible to compute the cost-optimal path between each pair of resource locations, we will require $\hat{E}$ to contain only a minimal set of edges by removing transitive connections. A transitive connection is a path within the road network that contains at least one intermediate resource location. For example, if along the cost-optimal path from X^A to X^B resource location X^C is encountered, then X^A and X^B would not be connected in the resource graph (cf. Figure 1(b)). However, $\hat{G}$ would contain edges connecting X^A and X^C as well as X^C and X^B. We explicitly exclude transitive connections from the resource graph for two reasons. First, transitive



(a) Optimal direct paths between resources are marked green, fastest paths with intermediate resources are marked red.



(b) Edges of the resource graph (marked continuously green) and excluded transitive connections (marked dashed red).

Figure 1: Illustration of a query node q and resources A, B, C in a road network graph (a) and the respective resource graph (b).

links do not allow computing the success probability correctly because the intermediate resource locations are not considered. Second, the existence of transitive connections leads to the inefficient traversal of identical subpaths.

We also compute the cost-optimal paths from the query node q to all resource locations. As there is no gain in returning to the query node, paths ending at q are excluded from the computation. Algorithm 1 describes the computation of $\hat{E}$ which is illustrated in Figures 1. Note that in order to compute $\hat{E}$, we need to compute all cost-optimal paths within the road network graph and then prune the transitive connections. It is not possible to avoid the computation of transitive connections directly.

The cost function of the road network graph G naturally extends to $\hat{G}$. Since every edge in $\hat{G}$ corresponds to an cost-optimal path in G, the cost function $\hat{c} : \hat{E} \to \mathbb{R}_0^+$ maps an edge of $\hat{E}$ to the accumulated costs of the respective path in G. Analogously, $\hat{t} : \hat{E} \to \mathbb{R}_0^+$ maps an edge of $\hat{E}$ to the accumulated time of the respective cost-optimal path in G.

Combining the above, we define the resource graph as $\hat{G} = \hat{G}_{(q,\mathcal{X})} := (\hat{V}, \hat{E}, \hat{c}, \hat{t})$. Note that $\hat{G}$ holds all the query-relevant information since it contains the query node q as well as the re-

Algorithm 1: Computation of $\hat{E}$

```
Input: Query setting $\mathcal{X}(q), q$
Output: Edges $\hat{E}$ of resource graph $\hat{G}$
1  begin
2      $\hat{E} \leftarrow \emptyset$
3      foreach $X \in \mathcal{X}(q)$ do
4          Compute fastest path $p$ from $q$ to $X$
5          if no intermediate resource on $p$ then
6              $\hat{E} \leftarrow \hat{E} \cup p$
7          end
8          Compute multi-target Dijkstra from $X$ to all
           $X' \in \mathcal{X} \setminus X$ in $G$
9          foreach fastest path $p$ from $X$ to $X'$ do
10             if no intermediate resources on $p$ then
11                 $\hat{E} \leftarrow \hat{E} \cup p$
12             end
13         end
14     end
15 end
```

source locations $\mathcal{X}(q)$. Therefore, speaking of a query setting, we mean q and $\mathcal{X}(q)$ as well as the according resource graph $\hat{G}$.

4.2 Resource Routes and PRRQR

Relying on $\hat{G}$, we may now define the possible solutions to our query. For a given query setting $q, \mathcal{X}(q)$ and the according resource graph $\hat{G}$, a *resource route* is a route r in $\hat{G}$, starting at query node $q = X^0$. Note that by construction of the resource graph a resource route only follows optimal paths between resources. We describe a resource route as a set of edges (in $\hat{G}$), i.e., $r = (e_{r_1}, \dots, e_{r_n})$, where $e_{r_i} \in \hat{E}$ for all $1 \leq i \leq n$. Since in our context the order of resources is of particular interest, we introduce a specific notation for it. Recall that every edge in $\hat{E}$ connects two resources unless it starts at the query node. Hence, along a resource route r with n edges we encounter n resources. Note that these resources are not necessarily distinct, as r may contain cycles. Let $X_r = (X^{r_1}, \dots, X^{r_n})$ denote the n resources along r. To introduce a measure for the success probability of a complete route, we start by defining the success probability of a resource route.

Definition 1. *For a given resource route r along resources $(X^{r_1}, \dots, X^{r_n})$, let t_i denote the* time of arrival *at X^{r_i}, i.e., $t_0 < t_1 < \dots < t_n$, and let c_i denote the accumulated cost up to resource X^{r_i}. Furthermore, let $\{P(X^{r_i}, t), t \geq 0\}$ denote the transition matrices of X^{r_i} (dependent on the parameters of X^{r_i}) and let $\pi_0(X^{r_i})$ denote the initial distribution of the states of X^{r_i} (dependent on the availability of short term observations regarding X^{r_i}). Then the probability distribution of X^{r_i} at t_i is defined as:*

$$\big(\mathbb{P}(X_{t_i}^{r_i} = 0), \mathbb{P}(X_{t_i}^{r_i} = 1)\big) := \pi_{t_i}(X^{r_i}) = \pi_0(X^{r_i})P(t_i)$$

Hence, the success probability *of X^{r_i} at arrival time t_i is the probability that resource X^{r_i} is in state* `available` *at time t_i, i.e., $\mathbb{P}(X_{t_i}^{r_i} = 0)$.*

Note that in accordance with the probabilistic model as presented in Section 3, we denote state `available` of a resource X by $X = 0$ and the state `consumed` by $X = 1$. Based on this definition, we are able to define the success probability for a complete resource route:

Definition 2. *Let $q, \mathcal{X}(q), \hat{G}$ be a query setting and r be a resource route in $\hat{G}$. The* Success Probability *of r, denoted by $\mathbb{P}_S(r)$, is defined as the probability of the complementary event of not finding any available resource along r:*

$$\mathbb{P}_S(r) := 1 - \left(\prod_{i=1}^{n} \mathbb{P}(X_{t_i}^{r_i} = 1)\right)$$

Given the success probability, we now need to find a second measure which measures the effort of finding an available resource, i.e., the expected cost of a resource route:

Definition 3. *Let $q, \mathcal{X}(q), \hat{G}$ be a query setting and r be a resource route in $\hat{G}$. The* Expected Cost *of r, denoted by $\mathbb{E}_c(r)$, is defined as:*

$$\mathbb{E}_c(r) := \sum_{i=1}^{n}\left(c_i \cdot \mathbb{P}(X_{t_i}^{r_i} = 0) \cdot \prod_{j=1}^{i-1} \mathbb{P}(X_{t_j}^{r_j} = 1)\right)$$

Relying on both measures, we can now define the *Probabilistic Resource Route Query with Reappearance (PRRQR)*:

Definition 4. *Let q be a query node, $\mathcal{X}(q)$ the set of corresponding resources, $\hat{G}$ the according resource graph. Furthermore, let $0 \ll \rho < 1$ denote a probability threshold. The result of a PRRQR with threshold ρ, denoted by PRRQR(ρ), is the resource route in $\hat{G}$ with minimal expected cost among all resource routes which exceed the probability threshold ρ.*

$$PRRQR(\rho) = \arg\min\{\mathbb{E}_c(r) \mid \mathbb{P}_S(r) \geq \rho\}$$

Note that there exists a straightforward variation of the PRRQR. Instead of thresholding the success probability, one could bound the maximal cost. Given a cost threshold $\tau > 0$, the query result is the resource route maximizing the success probability while not exceeding the cost bound τ. In this case, it is usually more reasonable to employ τ as a strict bound on the maximal cost instead of bounding the the expected cost. For example, consider driving an electric vehicle with a limited remaining range. Bounding the expected distance misses the point, only bounding the actual driven distance (while maxmimizing the success probability) will provide a suitable route. This variation of the query is also examined our experiments. However, in the following, we focus on the first (and more sophisticated) case to keep the description compact.

5. QUERY PROCESSING

In this section, we present algorithms for processing PRRQRs. First, we propose two heuristics which are employed in a greedy search that computes approximations of the optimal results. Afterwards, we will present two methods computing optimal solutions, relying on backtracking as well as on branch and bound. In the following, let $0 < \rho < 1$ be a probability threshold, and let $\hat{G}$ be the resource graph according to a given query setting $q, \mathcal{X}(q)$.

Let us note some aspects central to the PRRQR before going into the details of the algorithms. Given $\hat{G}$ and the query position q, then all resource routes start at q by definition. Hence, the set of possible solutions may be conceived as a search tree rooted at q where each branch is a sequence of resource locations. For a probability-constrained PRRQR with $\rho = 1$, i.e., a PRRQR requiring certainty of finding a resource, this search tree is infinite. The effect is caused by the time-dependent decay inherent in our model. Thus, a certain observation of availability of a particular resource will no longer be certain at the time of arrival. As a result, the success probability of a resource route can only asymptotically converge against 1. Even when $\rho < 1$, the search space is generally very large. This is because considering resource reappearance adds considerably to the complexity of the task. Similar to the Traveling Salesman Problem (TSP), there is no local optimality w.r.t. the resource subsequences that can be exploited. Hence, it is not possible to tell whether a resource route r can be extended into an optimal solution based on its current $\mathbb{P}_S(r)$ and $\mathbb{E}_c(r)$. Furthermore, it is easy to see that all permutations of the set of resources can be found in the search tree. Thus, from a theoretic point of view the problem is NP-complete. Only by setting the threshold $\rho < 1$, the search space becomes finite.

One precomputational step which all algorithms have in common is the computation of the resource graph $\hat{G}$ and its edges which constitute cost-optimal paths between resource locations in the underlying road network graph. The pseudocode for this operation is given in Algorithm 1. The set of edges $\hat{E}$ is realized as an adjacency matrix A of dimension $N + 1 \times N$, where $|\mathcal{X}(q)| = N$ is the number of suitable resources of the respective query setting. For notational reasons, we denote the query node q by X^0. The entries $a_{ij}, 0 \leq i \leq N, 1 \leq j \leq N$ of A are defined as:

$$a_{ij} = \begin{cases} c(p(X^i, X^j)) & \nexists X^k \in p(X^i, X^j), i, j, k \text{ pairw. inequal} \\ \infty & else \end{cases}$$

where $p(X^i, X^j)$ denotes the cost-optimal path from X^i to X^j within the underlying road network graph. A holds the cost of all cost-optimal paths, both pairwise between resources as well as from q to any resource, if no intermediate resources are located along this path. A, however, does not hold any information about paths from any resource to q, because the query node is not a resource and thus does not yield any gain toward the query goal. In case cost does not refer to travel time, we also compute the travel times of the cost-optimal paths. Recall the example of a query setting and its resource graph, as illustrated in Figure 1.

The according adjacency matrix of this scenario is given by:

	A	B	C
q	∞	$t(p(q, B))$	$t(p(q, C))$
A	∞	∞	$t(p(A, C))$
B	∞	∞	$t(p(B, C))$
C	$t(p(C, A))$	$t(p(C, B))$	∞

As mentioned before, depending on the application, the subset of suitable resource locations may be query-dependent. However, as the total set of resource locations is known prior to the query, it is possible to precompute the above adjacency matrix. If at query time a selection of suitable resources is required, the non-relevant rows and columns may simply be ignored. In the following, we consider an appropriate adjacency matrix as given. This assumption is not to the disadvantage of any of the proposed algorithms, as they all rely on the resource graph $\hat{G}$ and its edges modeled by the adjancency matrix.

5.1 Heuristic Solutions

In order to cope with the complexity of the PRRQR, we propose two search heuristics employed in greedy algorithms. Both heuristics aim at exceeding the given probability threshold by extending a (partial) resource route r by the “best” next resource location. The first heuristic greedily chooses the resource location which yields the best success probability upon arrival, while the second heuristic greedily chooses the resource location which yields the best tradeoff between success probability upon arrival and cost to reach the location from the present one. Formally, we propose to evaluate a possible extension of resource route r along resources $(X^{r_1}, \dots, X^{r_{i-1}})$ by one of the resource locations $\{X^1, \dots, X^N\}$ according to the following heuristics:

(1) Extend r by X^{r_i} such that

$$X^{r_i} = \arg\max\{\mathbb{P}(X^{r_j}_{t_j} = 0) \mid 1 \leq j \leq N\}$$

(2) Extend r by X^{r_i} such that

$$X^{r_i} = \arg\max\{\mathbb{P}(X^{r_j}_{t_j} = 0)/\hat{c}(e_{r_j}) \mid 1 \leq j \leq N\}$$

where $\hat{c}(e_{r_j})$ denotes the cost for traveling from resource location $X^{r_{j-1}}$ to X^{r_j} along the an cost-optimal path.

In conclusion, our greedy approaches **G1** and **G2** proceed as follows: For a given query setting $q, \mathcal{X}(q)$ and a probability threshold $0 < \rho < 1$ all cost-optimal paths from q to all resource locations adjacent to q w.r.t. the adjacency matrix are computed. Then, **G1** and **G2** choose the most promising extension according to the extension strategies (1) and (2), respectively. If the success probability of the obtained resource route does not exceed the probability threshold ρ, the procedure is repeated for all resource locations adjacent to the current one w.r.t. the adjacency matrix. As soon as the success probability exceeds ρ, we have found a viable solution.

5.2 Optimal Results

The greedy approach described above aims at the computation of reasonable resource routes in efficient time. However, in some applications, quality is more important than efficiency. For these cases, we propose two different approaches which guarantee optimal results. We present a backtracking algorithm and a further accelerated branch and bound approach.

5.2.1 Optimal Results through Backtracking

The backtracking approach, denoted by **BT**, starts at query node q and gradually expands resource routes as long as they qualify as result candidates. A resource route disqualifies as a result candidate if it exceeds the expected cost of the currently best resource route. During the expansion, **BT** explores the search tree (rooted at q) in depth-first order. Note that this search tree is generally infinite. Consequently, it is of even greater importance to exclude resource routes from expansion early on in the algorithm. Therefore, we conduct a prior initialization step equal to an execution of the greedy **G2** algorithm. This generates a valid resource route in efficient time, its expected cost may be used as a first bound. We omit the initialization step here (since it is equal to the description of **G2** above). Instead, we only give the recursive procedure as presented in Algorithm 2.

The procedure `expandRecursive` is initially called with a trivial resource route only consisting of query node q and an unspecified resource (which is reset to an adjacent resource during the first run). The expected cost of the result generated by **G2** is held in a global variable M_c as an initial upper bound for the expected cost. While traversing the search tree M_c will be tightened by finding better solutions. In line 3, candidates which do not qualify as results are excluded, while in line 6 possible result are generated (i.e., resource routes exceeding the probability threshold). The actual search tree traversal is realized recursively in lines 10, 11. If an expansion r' of a resource route r is better than the current best route along this subtree $\hat{r}$ (w.r.t. the expected cost), then $\hat{r}$ is updated to r' and M_c is updated to $\mathbb{E}_c(r')$ (lines 13,14). Thus, by sequential traversal of the search tree, **BT** returns the optimal result upon termination. However, due to the exponential number of

Algorithm 2: Expansion step of **BT**

```
1  expandRecursive(Resource route $r$,
2  resource $X$)
   Data: Upper bound for $\mathbb{E}_c$, $M_c$
   Output: Optimal resource route $\hat{r}$
3  begin
4      if $\mathbb{E}_c(r) > M_c$ then
5          return $\emptyset$
6      end
7      if $\mathbb{P}_S(r) > \rho$ then
8          return $r$
9      end
10     initialize variable holding current best route $\hat{r} = \emptyset$
11     foreach resource $X$ adjacent to last resource of $r$ do
12         $r' \leftarrow$ expand($r, X$)
13         if $\hat{r} = \emptyset \vee \mathbb{E}_c(r') < M_c$ then
14             $\hat{r} = r'$
15             $M_c \leftarrow \mathbb{E}_c(\hat{r})$
16         end
17     end
18     return $\hat{r}$
19 end
```

branches and the technically infinite length of the branches runtime is prone to degenerate. Therefore, we propose another algorithm which computes optimal results in significantly less time.

5.2.2 *Optimal Results through Branch and Bound*

Like the backtracking algorithm **BT**, this branch and bound approach, denoted by **BB**, relies on an upper bound for the expected cost (M_c) which is tightened as the algorithm progresses. Additionally, **BB** incorporates a forward estimation for the expected cost a route minimally needs to exceed the probability threshold ρ. The forward estimation is a lower bound for the expected cost w.r.t. a resource route and ρ. Consequently, if this lower bound exceeds the upper bound for the expected cost M_c, r can be excluded from further expansion, i.e. the respective subtree can be pruned.

Algorithmically, **BB** is similar to **BT**, except for the mentioned forward estimation. This forward estimation is incorporated into Algorithm 2 as an if($m_c < M_c$))-statement spanning from line 11 through line 17, where m_c is the output of procedure `forwardEstimation`, as presented in Algorithm 3. Before each possible expansion of a resource route r, `forwardEstimation` is called with r and the probability threshold ρ. In lines 3-7 the parameters are set which are subsequently used to compute the lower bound for the expected travel time. t_{now} is the absolute travel time of input resource route r. t_{min} is the fastest travel time between any two resources. Thus, t_{opt} is the minimal possible arrival time at the next resource w.r.t. the absolute travel time of r. p_{opt} and m_c are initialized with $\mathbb{P}_S(r)$ and $\mathbb{E}_c(r)$, respectively. Both values are updated in the while loop (lines 8-13) until $p_{\text{opt}} \geq \rho$, i.e. until the optimal success probability exceeds the threshold. Now, let us investigate the operations in the while loop. First, p_{max} is defined as the maximal probability among all resources at the minimal possible arrival time t_{opt}. Note that we only allow observations to be incorporated into the model until query time. Therefore, p_{max} is monotonically decreasing in t_{opt}, and it converges against the minimal value of all stationary distributions in state `consumed`. m_c is extended by a new summand reflecting a hypothetical and optimal journey to the resource with maximal probability and minimal cost. Consequently, the success probability bound is updated to the probability of the complementary event of not finding any available resource along this optimal journey. By this strategy, in every iteration of the while loop, a journey to an optimal next resource causing minimal cost is simulated. Thus, the maximal success probability is aggregated while assuming minimal cost. We prove this in the following lemmas. We introduce the following terminology: For a given resource route *r* we refer to any iteration of the while loop of Algorithm 3 as an *optimal extension*. This coincides with the above described intuition.

Algorithm 3: Forward Estimation of **BB**

```
1  forwardEstimation(Resource route $r$,
   current $\mathbb{E}_c$ bound $M_c$)
   Output: Upper bound for the success probability
           of any extension of $r$ until it exceeds $M_c$
2  begin
3      $t_{\text{now}} \leftarrow$ arrival time at last resource of $r$
4      $t_{\text{min}} \leftarrow \min_{e \in \hat{E}} \hat{t}(e)$
5      $t_{\text{opt}} \leftarrow t_{\text{now}} + t_{\text{min}}$
6      $p_{\text{opt}} \leftarrow \mathbb{P}_S(r)$
7      $m_c \leftarrow \mathbb{E}_c(r)$
8      while $m_c < M_c$ do
9          $p_{\text{max}} \leftarrow \max_{X \in \mathcal{X}(q)} \mathbb{P}(X_{t_{\text{opt}}} = 0)$
10         $m_c \leftarrow m_c + \big(t_{\text{opt}} \cdot p_{\text{max}}(1 - p_{\text{opt}})\big)$
11         $p_{\text{opt}} \leftarrow 1 - \big((1 - p_{\text{opt}})(1 - p_{\text{max}})\big)$
12         $t_{\text{opt}} \leftarrow t_{\text{opt}} + t_{\text{min}}$
13     end
14     return $p_{\text{opt}}$
15 end
```

LEMMA 1. *In any optimal extension the gain of the updated values $m_c' \leftarrow m_c$ and $p_{opt}' \leftarrow p_{opt}$ yield the best possible trade-off between expected cost and success probability. More specifically, let r be a resource route with $\mathbb{E}_c(r) = m_c$, $\mathbb{P}_S(r) = p_{opt}$ and travel time t_{opt}. Then the following statement holds: For any possible extension r′ of r to another resource:*

$$\frac{m_c' - m_c}{p_{opt}' - p_{opt}} < \frac{\mathbb{E}_c(r') - \mathbb{E}_c(r)}{\mathbb{P}_S(r') - \mathbb{P}_S(r)}$$

PROOF. *In order to prove this lemma, we need to formulate the success probability of a resource route r differently:*

$$\begin{aligned}\mathbb{P}_S(r) &= 1 - \prod_{i=1}^{n} \mathbb{P}(X_{t_i}^{r_i} = 1) \\ &= \sum_{i=1}^{n} \left(\mathbb{P}(X_{t_i}^{r_i} = 0) \cdot \prod_{j=1}^{i-1} \mathbb{P}(X_{t_j}^{r_j} = 1) \right)\end{aligned}$$

Note that the equality indeed holds. This is because the event of finding at least one available resource can be described by the complementary event of not finding any resource in state available. Equally, it can be described as the union of events that resource X^{r_i} is available but all other resources thus far were consumed. Now, for a given resource route r, let r′ denote an arbitrary extension of r by another resource X with respective arrival time t. By the alternative definition of $\mathbb{P}_S$, we have $\mathbb{P}_S(r') = \mathbb{P}_S(r) + t\mathbb{P}(X_t = 0) \cdot (1 - \mathbb{P}_S(r))$. Recall that $\mathbb{E}_c(r) = m_c$ and $\mathbb{P}_S(r) = p_{opt}$.

Now, we show our claim:

$$\frac{m'_c - m_c}{\mathbb{E}_c(r') - m_c} < \frac{p'_{opt} - p_{opt}}{\mathbb{P}_S(r') - p_{opt}}$$

$$\frac{c'_{opt}p'_{opt}(1 - \mathbb{P}_S(r))}{c(r)\mathbb{P}(X_t = 0)(1 - \mathbb{P}_S(r))} < \frac{p'_{opt}(1 - \mathbb{P}_S(r)}{\mathbb{P}(X_t = 0)(1 - \mathbb{P}_S(r)}$$

By definition, $t'_{opt} \leftarrow t_{opt} + t_{min}$, where t_{min} denotes the minimal travel time in $\hat{E}$. Consequently, $t'_{opt} < t$, therefore, the inequality holds which proves the claim. □

LEMMA 2. *Let r be a resource route. The forward estimation of the expected cost as computed by Algorithm 3 is indeed a lower bound.*

PROOF. *This follows from the following properties:*

(i) The number of optimal extensions needed until r exceeds the probability threshold is at most the number of actual extensions needed.

(ii) Every optimal extension of r yields a better trade-off than an actual extension.

(iii) No sequence of actual extensions of r can exceed the probability threshold while yielding a lower expected cost than the sequence of optimal extensions chosen by Algorithm 3.

(i) follows directly from the definition of $p_{opt} \leftarrow 1 - \big((1 - p_{opt})(1 - p_{max})\big)$. In every optimal extension, p_{opt} is increased by the maximally possible value. Therefore, no other sequence of extensions can yield a faster increase. (ii) is the statement of Lemma 1. Finally, (iii) follows from both, (i) and (ii). □

Note that all of the above is easily applied to the case where instead of minimizing the expected cost w.r.t. a probability threshold we maximize the probability w.r.t. an absolute cost bound (as introduced at the end of Section 4.2). For example, consider the backtracking expansion Algorithm 2. Instead of dismissing (storing) a route r if $\mathbb{E}_c(r) > M_c$ ("$<$" holds), in the complementary scenario, a route r is dismissed (stored), if $c(r) > M_c$ ("$<$" holds). Similarly for the forward estimation presented in Algorithm 3. Again, the expected cost $\mathbb{E}_c(r)$ is to be replaced with the absolute cost $c(r)$. While the absolute cost bound is not exceeded, optimal path extensions are simulated, adding maximal success probability to the path. When the cost bound is exceeded and the maximal current best success probability is not surpassed, the search tree can be pruned. If, on the other hand, the success probability is surpassed by the optimal path extension, the path (and its subtree in the search tree) qualifies as a candidate. As for the theoretic arguments, they apply analoguously, therefore we omit an adapted version due to space limitations.

Concludingly, we have presented four algorithms for solving the proposed PRRQR in this section. **G1** and **G2** follow a greedy heuristic to produce approximate results, while **BT** and **BB** produce exact results. **BB** is an extension of **BT** which makes use of a lower bound forward estimation of the expected cost. In the above lemmas, we have shown correctness of the proposed bound.

6. EXPERIMENTAL EVALUATION

We evaluate our model and our algorithms on settings in real world road networks extracted from OpenStreetMap[1] (OSM) using the MARiO framework [6]. All experiments were conducted on a desktop computer equipped with an Intel Core i7-3770 CPU and 32 GB RAM, running Java 1.64 (64-Bit) on Linux 3.13 x86_64. Different algorithms are always tested on the same randomly generated scenario before comparing results. Runtime evaluations are based on Java's nanotime clock and performed for each algorithm individually excluding preliminary steps like graph population and building of the adjacency matrix. Computation of the latter takes around 250 milliseconds, for standard settings in the *Parking* and *Charging* scenarios, respectively. Note that all cost-optimal paths were computed using Dijkstra's algorithm. Choosing a different routing algorithm and/or employing a speed-up technique would yield the same benefit for all compared approaches. Modifying the path computation algorithm is an easy task, however, on a city scale (which the applications require) this would hardly yield any computational benefit. We present experiments for two realistic applications:

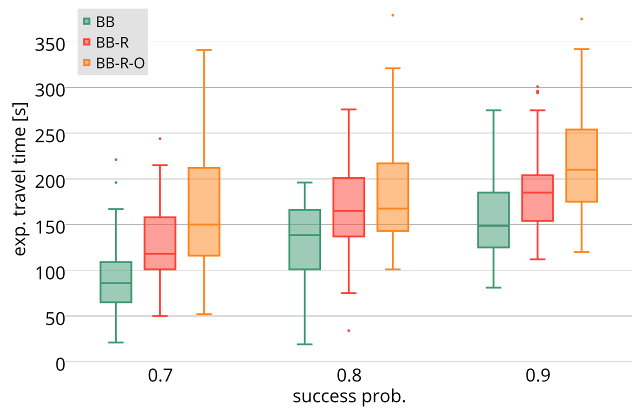
- *Parking* scenario (located in Bamberg, Germany): Given a probability threshold, we provide a route along parking spots which surpasses the threshold and minimizes the expected travel time. This scenario is based on ground truth extracted from OSM metadata.
- *Charging* scenario (located in Brussels, Belgium): Given a query position and a range limit (as used by electric vehicles), we provide a route along charging stations not exceeding the range limit and maximizing the success probability.

Note that these scenarios are complementary w.r.t. the criterion which is bounded and the criterion which is to be optimized, as explained in Section 4.1. While *Charging* relies on an hard numeric bound (remaining range), *Parking* relies on the more sophisticated expected value bound. Therefore we choose *Parking* as our main scenario. We will not present all charts for both scenarios, however noting that corresponding charts show the same behavior.

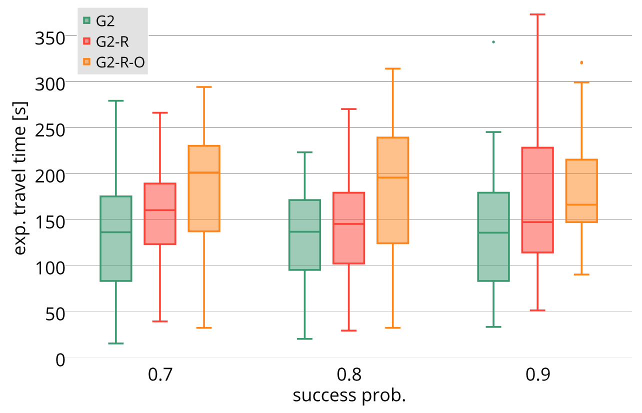
6.1 Parking Scenario

We generated the following test cases on the road network of the city of Bamberg, Germany, containing approximately 10.000 nodes and 20.000 edges as well as nearly exhaustive metadata regarding parking spots. For every test case, a target node is randomly drawn from all road network nodes of degree ≥ 1 within a three kilometer radius from the city center. Then, an isochrone of 800 meters walking distance is computed around the target. Let N be the number of resources (according to the ground truth) within the isochrone. In our experiments, resources are rather dense, i.e., $25 \leq N \leq 100$. Subsequently, the query node q is randomly drawn from all nodes within the isochrone. This corresponds to the use case where we expect the user to trigger the query when they are in the vicinity of their target. The average and maximal distance from q to a resource are by construction 800 and 1600 meters, respectively. Finally, $M \leq N$ observations of resource availability are randomly distributed among the resource locations, and the respective sojourn times in the states `available` and `consumed` are set. For reasons of clarity, in our experimental settings the sojourn times are set to the same configurations for all resources. We assume the expected time a spot stays vacant (`available`) to be 3 minutes and the expected time a spot stays occupied (`consumed`) to be 90 minutes. Note that the resources could easily be parametrized separately to model differently volatile resources. In this scenario, a probability threshold is given, and as a cost function we use travel time as formalized in Definition 3. The optimal resource route is the one with the least expected travel time among all resource routes with a success probability exceeding the threshold.

[1]http://www.openstreetmap.org/

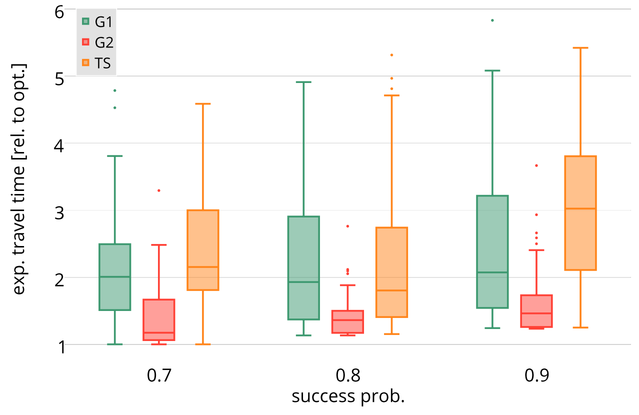


(a) **BB**-related algorithms

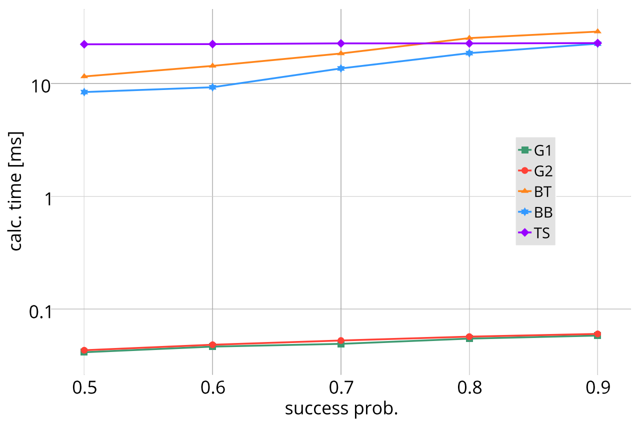


(b) **G2**-related algorithms

Figure 2: Illustration of the influence of model complexity on the quality of results (*Parking* scenario.



(a) Expected travel time relative to optimal solution



(b) Calculation time

Figure 3: Illustration of quality as well as efficiency of all algorithms in the *Parking* scenario.

First, we want to evaluate how much the additional information held by our probabilistic model improves result quality. Recall that our model supports reappearance and incorporates short term observations, two properties that distinguish this work from others. In order to prove that the gain in result quality outweighs the gain in model complexity, we trim our algorithms **BB** (branch and bound) and **G2** (greedy approach with probability per cost heuristic) to partially ignore the information provided by the underlying model. In a first step, we disable the possibility of resource reappearance, we denote these approaches by **BB-R** and **G2-R**, respectively. This means, **BB-R** and **G2-R** proceed like their respective counterparts but do not revisit resources which have previously been observed as `consumed`. This corresponds to a simpler probabilistic model without the feature of resource reappearance. In a second step, we additionally disable short term observations. We denote these approaches by **BB-R-O** and **G2-R-O**. They proceed like **BB-R** and **G2-R**, respectively, but additionally ignore any short term observations. Hence, the variations emulate an even simpler model which only allows static uncertainty, as used in [4], for example.

The results for the **BB**-related and the **G2**-related algorithms are shown in Figures 2(a) and 2(b), respectively. Both figures depict the same settings. It is obvious that requiring a greater probability threshold results in resource routes with longer expected travel time. Therefore, the overall increase in expected travel time is consequential. Both figures clearly show that the algorithms which rely on greater information, i.e., use a more complex model, yield better results. Figure 2(a) visualizes the results of the branch and bound approaches which are optimal w.r.t. to the information available. As claimed, **BB** on average outperforms **BB-R**, its counterpart which does not allow reappearance by at least 20 percent. **BB-R**, in turn, outperforms its counterpart which does not incorporate short term observations, **BB-R-O**. This supports the previously made claim that resource reappearance and short term observations do indeed improve the quality of results. From Figure 2(b) we observe, that simpler algorithms also benefit from the additional information contained in the model. Comparing the two figures, **BB**-algorithms of course yield better results than **G2**-algorithms and do so with significantly less variance than the greedy approaches. This is because the heuristics rely on chance in the form of beneficial problem settings in order to generate near-optimal results.

Next, let us investigate the performance of the algorithms presented. As mentioned before, there exists no work which is fully comparable. However, as the PRRQR is related to the TSP and clustering is commonly used to approximate the TSP (as in [3]), we use this concept to implement an approximative comparison partner denoted by **TS**. It is important to mention that **TS** does not support resource reappearance, because otherwise the heuristic would not visit sufficiently many distinct resources to achieve a compara-

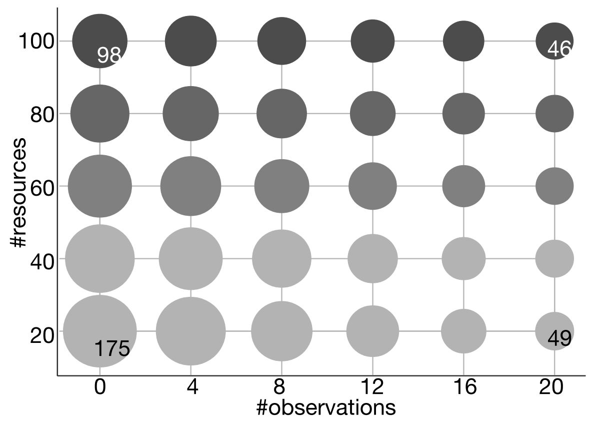
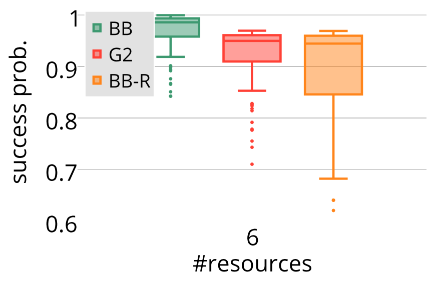


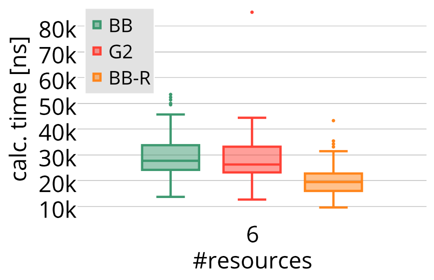
Figure 4: Influence of the number of resources and the number of observations on the expected travel time, i.e., result quality (for a probability threshold of 0.7). (The gray shades only serve an aesthetic purpose.)

ble success probability. Before we present the results, let us explain how **TS** proceeds. In a first step, **TS** conducts a k-medoid clustering on the set of all resources, where experimentally $k = 6$ has proven adequate. Subsequently, a TSP on the cluster medoids (starting at the query node) is solved. Then follows the actual resource route computation. It starts at the query node and computes the cost-optimal path to the first medoid. In the respective cluster, a greedy depth-first search (starting at the medoid) is conducted, returning an approximation of the cluster-internal cost-optimal path. From the last resource of the cluster we compute the cost-optimal path to the next medoid. This procedure is continued until the resource route exceeds the given probability threshold. **TS** serves as an algorithmic competitor based on a simpler probabilistic model but with a solid heuristic that has proven efficient when solving TSP-related problems. Note that the cost-optimal paths between all resources are precomputed in order to make the comparison to our algorithms – which use the precomputed adjacency matrix – fair.

We compare **TS** to all algorithms introduced in Section 5, i.e., the two greedy approaches **G1** and **G2** as well as the exact solutions **BT** and **BB**. Figure 3(a) shows the quality of the results produced by the approximative algorithms, i.e., **G1**, **G2**, and **TS**. Their respective expected travel times are given relative to the optimal results. The higher the probability threshold, i.e., the more complex the task, the greater the discrepancy between optimal results and approximation. Although **G2** relies on the rather simple probability-to-cost ratio heuristic, it significantly outperforms its comparison partners. While **G2** yields near-optimal results in the easier settings, the optimal solutions in the most elaborate scenario (probability threshold 0.9) undercut its expected travel times on average by about 30 percent. This gain in quality, however, comes at the price of calculation time, as depicted in Figure 3(b). This illustration shows the averaged runtimes of all algorithms when increasing the required probability threshold. The greedy approaches generate results in almost interactive time, while **BB**, **BT**, and **TS** are around two to three orders of magnitude slower. However, it is important to note that two orders of magnitude only correspond to around 100 ms of calculation time. Comparing the exact algorithms, we observe that **BB** outperforms **BT** which can be attributed to the forward estimation. The competitive approach **TS** performs in constant time of about 150 milliseconds (for the same number of resources), however generating the worst results.



(a) Success probability



(b) Calculation time

Figure 5: Illustration of quality as well as efficiency of selected algorithms in the *Charging* scenario.

Finally, we want to explore how volatile the results are w.r.t. the model parameters. We restrict ourselves to the optimal solution provided by **BB**, seeing as the quality ratio of optimal to approximative solutions has been explored above. Figure 4 depicts the influence of the number of parking spots relative to the number of short term observations of vacant parking spots. Thus, each circle in the plot corresponds to a pair of parameter values, and the diameter of each circle represents the average expected travel time of this scenario in seconds, as do the numbers in the corner circles. The result shows the expected behavior that with an increasing number of parking spots, expected travel time decreases. Furthermore, for any given scenario, it can be seen that the increased amount of short term observations also reduces the expected time until a vacant spot is found. Similarly expectable behavior is observed when varying the sojourn time parameters $1/\lambda$ and $1/\mu$, therefore further charts are omitted.

6.2 Charging Scenario

For *Charging* we generated test cases on the road network of the city of Brussels, Belgium, containing approximately 30.000 nodes and 67.000 edges. For every test case a query node is randomly drawn from all road network nodes of degree ≥ 1 within a 6 kilometer radius of the city center. Then, an isochrone of 6 kilometers is computed around the query node, wherein 6 resource locations are randomly drawn. We have evaluated other numbers of resources but the results do not reveal additional information and are therefore omitted here. Compared to *Parking*, where nearly every street holds at least one resource, this scenario models resource scarcity. Again, if N denotes the number of resources (6 in our experiments), then $M \leq N$ observations of resource availability are randomly distributed among these resource locations. The expected time a charging station remains vacant (`available`) is set to 30 min, and the expected time it remains occupied (`consumed`) is set to 50 minutes. As before, every charging station may be parametrized individually, however we pass on it for reasons of clarity and lack of ground truth. In this scenario an absolute distance bound of 6 kilometers is given, emulating the remaining range of an electric vehicle with low battery. Note that in contrast to *Parking*, this bound is strict and cannot be exceeded. Every algorithm computes a route with an absolute distance of 6 kilometers, the optimal resource route is the one with maximal success probability.

In a first setting, we compare the result quality of our exact algorithm **BB**, our greedy solution **G2** and **BB-R** (cf. Figure 5(a)), the branch and bound variation which does not incorporate resource reappearance. Additionally to the scarcity of resources, the remaining range (6 kilometers) is only double the average distance from query node to the next resource (3 kilometers, as resources are distributed uniformly within the isochrone). Due to these tightened constraints, superiority of the optimal results generated by **BB** becomes more apparent. In almost three out of four runs, **BB** yields a success probability of over 95 percent, outperforming **G2** significantly. While the greedy heuristic worked well before, it is now easily lead down a considerably less beneficial branch of the search tree. Nonetheless, **G2** still produces slightly better results than **BB-R**. Again, this advocates our model which supports resource reappearance. Even an approximative approach on our model yields better results than an exact algorithms on a less sophisticated model due to lack of information. Of course, a simpler model needs less intricate function evaluations. In our case, however, the difference is merely a matter of microseconds, as depicted in Figure 5(b).

Concludingly, we have empiricially proven the benefit of our probabilistic model. It improves the quality of results by incorporating richer information, especially for complex but also for simpler tasks while not causing any significant computational overhead. On the contrary, our greedy approaches deliver competitive results in near-interactive time while our branch and bound approach yields optimal solutions in efficient time.

7. SUMMARY AND OUTLOOK

In this paper, we investigate probabilistic route queries in road networks where the user is guided along a set of resources in order to maximize the probability of encountering an available resource. We aim to find a route with minimal expected cost among all routes exceeding a given probability threshold. We propose a novel framework in which resources are modeled as continuous-time Markov chains with two states, `available` and `consumed`. In contrast to similar problems, our framework allows for consumed resources to reappear and takes short term as well as long term observations into account. The introduced query, referred to as PRRQR, is theoretically NP-complete and has an unlimited search space.

To solve this problem, we propose approximative as well as optimal solutions. We employ two different search heuristics in a greedy algorithm to achieve a trade-off between accuracy and calculation time. Furthermore, solutions using backtracking and a branch and bound approach provide optimal solutions in competitive time. We demonstrate the superiority of our model as well as the efficiency and effectiveness of our algorithms on two realistic applications. The first is the search of a vacant parking spot, and the second is the search for a vacant charging station for electric vehicles.

For future work, we want to turn to settings considering other types of observations like competing drivers looking for the same type of resource. Furthermore, we want to investigate the influence of edge costs which might change during the search.

8. REFERENCES

[1] H. Chen, W.-S. Ku, M.-T. Sun, and R. Zimmermann. The multi-rule partial sequenced route query. In *Proceedings of the 16th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (ACMGIS'08)*, pages 10:1–10:10, 2008.

[2] D. Cheng, M. Hadjieleftheriou, G. Kollios, F. Li, and S.-H. Teng. On trip planning queries in spatial databases. In *Proceedings of the 9th International Conference on Advances in Spatial and Temporal Databases (SSTD'05)*, pages 273–290, 2005.

[3] A. Delis, V. Efstathiou, and V. Verroios. Reaching available public parking spaces in urban environments using ad hoc networking. In *Proceedings of the 12th International Conference on Mobile Data Management (MDM'11)*, pages 141–151, 2011.

[4] N. Dolev, Y. Doytsher, Y. Kanza, E. Safra, and Y. Sagiv. Computing a k-route over uncertain geographical data. In *Proceedings of the 10th International Conference on Advances in Spatial and Temporal Databases (SSTD'07)*, pages 276–293, 2007.

[5] G. Gidofalvi, T. B. Pedersen, T. Risch, and E. Zeitler. Highly scalable trip grouping for large-scale collective transportation systems. In *Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '08, pages 678–689, New York, NY, USA, 2008. ACM.

[6] F. Graf, H.-P. Kriegel, M. Renz, and M. Schubert. MARiO: Multi attribute routing in open street map. In *Proceedings of the 12th International Conference on Advances in Spatial and Temporal Databases (SSTD'11)*, 2011.

[7] Y. Kanza, R. Levin, E. Safra, and Y. Sagiv. Interactive route search in the presence of order constraints. *The International Journal on Very Large Data Bases (VLDB'10)*, pages 117–128, 2010.

[8] Y. Kanza, E. Safra, and Y. Sagiv. Route search over probabilistic geospatial data. In *Proceedings of the 11th International Symposium on Advances in Spatial and Temporal Databases (SSTD'09)*, pages 153–170, 2009.

[9] M. Kolahdouzan, C. Shahabi, and M. Sharifzadeh. The optimal sequenced route query. *The International Journal on Very Large Data Bases (VLDB'08)*, 17(4):765–787, 2008.

[10] R. Levin, Y. Kanza, E. Safra, and Y. Sagiv. An interactive approach to route search. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (ACMGIS'09)*, pages 408–411, 2009.

[11] S. Ma, O. Wolfson, and Y. Zheng. T-share: A large-scale dynamic taxi ridesharing service. In *Proceeding of International Conference on Data Engineering (ICDE'13)*, pages 410 – 421, 2013.

[12] L. Matos, J. Nune, and A. Trigo. Taxi pick-ups route optimization using genetic algorithms. In *Proceedings of the 10th International Conference on Adaptive and Natural Computing Algorithms (ICANNGA'11)*, pages 410–419, 2011.

[13] J. Norris. *Markov Chains*. Cambridge Univ. Press, Cambridge, 1998.

[14] D. O. Santos and E. C. Xavier. Dynamic taxi and ridesharing: A framework and heuristics for the optimization problem. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI'13)*, pages 2885–2891, 2013.

Cost Estimation of Spatial k-Nearest-Neighbor Operators*

Ahmed M. Aly
Purdue University
West Lafayette, IN
aaly@cs.purdue.edu

Walid G. Aref
Purdue University
West Lafayette, IN
aref@cs.purdue.edu

Mourad Ouzzani
Qatar Computing Research Institute
Doha, Qatar
mouzzani@qf.org.qa

ABSTRACT

Advances in geo-sensing technology have led to an unprecedented spread of location-aware devices. In turn, this has resulted into a plethora of location-based services in which huge amounts of spatial data need to be efficiently consumed by spatial query processors. For a spatial query processor to properly choose among the various query processing strategies, the cost of the spatial operators has to be estimated. In this paper, we study the problem of estimating the cost of the spatial k-nearest-neighbor (k-NN, for short) operators, namely, k-NN-Select and k-NN-Join. Given a query that has a k-NN operator, the objective is to estimate the number of blocks that are going to be scanned during the processing of this operator. Estimating the cost of a k-NN operator is challenging for several reasons. For instance, the cost of a k-NN-Select operator is directly affected by the value of k, the location of the query focal point, and the distribution of the data. Hence, a cost model that captures these factors is relatively hard to realize. This paper introduces cost estimation techniques that maintain a compact set of *catalog* information that can be kept in main-memory to enable fast estimation via lookups. A detailed study of the performance and accuracy trade-off of each proposed technique is presented. Experimental results using real spatial datasets from OpenStreetMap demonstrate the robustness of the proposed estimation techniques.

1. INTRODUCTION

The ubiquity of location-aware devices, e.g., smartphones and GPS-devices, has led to a variety of location-based services in which large amounts of geo-tagged information are created every day. This demands spatial query processors that can efficiently process spatial queries of various complexities. One class of operations that arises frequently in practice is the class of spatial k-NN operations. Examples of spatial k-NN operations include: (i) Find the k-closest hotels to my location (a k-NN-Select), and (ii) Find for each school the k-closest hospitals (a k-NN-Join).

The k-NN-Select and k-NN-Join operators can be used along with other spatial or relational operators in the same query. In this case, various query-execution-plans (QEPs, for short) for the same query are possible, but with some of the QEPs having better execution times than the others. The role of a query optimizer is to arbitrate among the various QEPs and pick the one with the least processing cost. In this paper, we study the problem of estimating the cost of the k-NN-Select and k-NN-Join operators.

To demonstrate the importance of estimating the cost of these operators, consider the following example query: *'Find the k-closest restaurants to my location such that the price of the restaurant is within my budget'*. This query combines a spatial k-NN-Select with a relational select (price $\leq$ budget). There are two possible QEPs for executing this query: (i) Apply the relational select first, i.e., select the restaurants with price $\leq$ budget and then get the k-closest out of them, or (ii) Apply an incremental k-NN-Select (i.e., distance browsing [14]) and evaluate the relational select on the fly; execution should stop when k restaurants that qualify the relational predicate are retrieved. Clearly, the two QEPs can have different performance. Thus, it is essential to estimate the cost of each processing alternative in order to choose the cheaper QEP. Observe that distance browsing is also applicable to non-incremental k-NN-Select (i.e., to QEP(i)). [14] proves that for non-incremental k-NN-Select, the number of scanned blocks is optimal in distance browsing. Thus, in this paper, we model the cost of distance browsing being the state-of-the-art for k-NN-Select processing.

In addition to modeling the cost of the k-NN-Select, we study the cost of the k-NN-Join. The k-NN-Join is a practical spatial operation for many application scenarios. For example, consider the following query that combines a relational or a spatial predicate with a k-NN-Join predicate. Assume that a user wants to select for each hotel, its k-closest restaurants (k-NN-Join predicate) such that the restaurant/hotel's price is within the user's budget (relational predicate), or that the restaurant/hotel's location is within a certain downtown district (spatial range predicate). Clearly, estimating the cost of a k-NN-Join is important to decide the ordering of the relational, spatial, and k-NN operators in the QEP. A k-NN-Join can also be useful when multiple k-NN-Select queries are to be executed on the same dataset. To share the execution, exploit data locality and the similarities in the data access patterns, and avoid multiple yet unnecessary scans of the underlying data (e.g., as in [11]), all the query points are treated as an outer relation and processing is performed in a single k-NN-Join. In this paper, we introduce a cost model for locality-based k-NN-Join processing [22], which is the state-of-the-art in k-NN-Join processing.

While several research efforts (e.g., see [2, 3, 4, 5, 7, 15, 17, 18, 23]) estimate the selectivity and cost of the spatial join and range operators, they are not applicable to k-NN operators. For instance, the cost of a spatial range operator is relatively easy to estimate because the spatial region of the operator, in which the query answer

*This research was supported in part by National Science Foundation under Grants IIS 0916614, IIS 1117766, and IIS 0964639.

10.5441/002/edbt.2015.40

resides, is predefined and fixed in the query. In contrast, the spatial region that contains the k-nearest-neighbors of a query point, in the case of a k-NN-Select, or a point of the outer relation in the case of a k-NN-Join, is *variable* since it depends on the value of k, the location of the point, and the density of the data (i.e., its distribution). These three parameters render the problem of k-NN cost-estimation more challenging.

In this paper, we introduce the *Staircase* technique for estimating the cost of k-NN-Select. The Staircase technique distinguishes itself from existing techniques by the ability to quickly estimate the cost of any query using an $O(1)$ lookup. The main idea of the Staircase technique is to maintain a compact set of catalog information that summarize the cost. We perform various optimizations to limit the size of the catalog such that it can easily fit in main-memory. We empirically compare the performance of the Staircase technique against the state-of-the-art technique [24]. We show that the Staircase technique has better accuracy for spatial non-uniform data in the two-dimensional space while achieving orders-of-magnitude gain in query estimation time. Having a fast query execution time is vital for location-based services that serve multiple queries at very high rates, e.g., thousands of queries per second. Thus, estimating the cost needs to be extremely fast as it is a preliminary step before the query itself is executed.

In addition to estimating the cost of k-NN-Select, we introduce three new techniques for estimating the cost of k-NN-Join. Similarly to the Staircase technique, the proposed techniques employ a compact set of catalogs that summarize the cost and enable fast estimation. First, we present the *Block-Sample* as our baseline technique. Then, we introduce the *Catalog-Merge* technique that has better estimation time than the Block-Sample technique, but incurs relatively high storage overhead. Then, we introduce the *Virtual-Grid* technique that incurs less storage overhead than the Catalog-Merge technique. To the best of our knowledge, estimating the cost of k-NN-Join has not been addressed in previous work.

The contributions of this paper can be summarized as follows:

- We introduce the Staircase technique for estimating the k-NN-Select cost.
- We introduce three novel techniques for estimating the k-NN-Join cost, namely the Block-Sample, Catalog-Merge, and Virtual-Grid techniques.
- We conduct extensive experiments to study the performance and accuracy tradeoff that each of the proposed technique offers. Our experimental results demonstrate that:
  - the Staircase technique outperforms the techniques in [24] by two orders of magnitude in estimation time and by more than 10% in estimation accuracy,
  - the Catalog-Merge technique achieves an error ratio of less than 5% while keeping the estimation time below one microsecond, and
  - the Virtual-Grid technique achieves an error ratio of less than 20% while reducing the storage required to maintain the catalogs by an order of magnitude compared to the Catalog-Merge technique.

The rest of this paper proceeds as follows. Section 2 introduces some preliminaries and discusses the related work. Section 3 presents the Staircase technique for estimating the cost of k-NN-Select. Section 4 presents the Block-Sample, Catalog-Merge, and Virtual-Grid techniques for estimating the cost of k-NN-Join. Section 5 provides an experimental study of the performance of the proposed techniques. Section 6 contains concluding remarks.

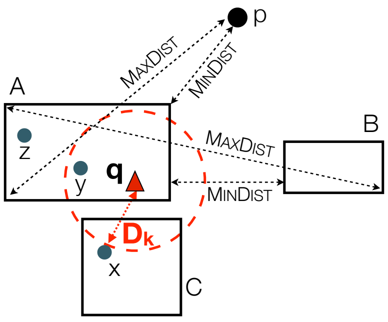


Figure 1: The MINDIST and MAXDIST metrics. In distance browsing [14], when $k = 2$ and q is a query focal point of a k-NN-Select, only blocks A and C are scanned, i.e., cost = 2.

2. PRELIMINARIES & RELATED WORK

We focus on the variants of the k-NN operations given below. Assume that we have two tables, say R and S, that represent two sets of points in the two-dimensional space. For simplicity, we use the Euclidean distance metric.

- ***k*-NN-Select**: Given a query-point q, $\sigma_{k,q}(R)$ returns the k-closest to q from the set of points in R.
- ***k*-NN-Join**: $R \bowtie_{kNN} S$ returns all the pairs (r, s), where $r \in R$, $s \in S$, and s is among the k-closest points to r.

Observe that the k-NN-Join is an asymmetric operation, i.e., the two expressions: $(R \bowtie_{kNN} S)$ and $(S \bowtie_{kNN} R)$ are not equivalent. In the expression $(R \bowtie_{kNN} S)$, we refer to Relations R and S as the outer and inner relations, respectively.

We assume that the data points are organized in a spatial index structure. However, we do not assume a specific indexing structure; our proposed techniques can be applied to a quadtree, an R-tree, or any of their variants, e.g. [12, 20, 13, 6, 16]. These are hierarchical spatial structures that recursively divide the underlying space/points into blocks until the number of points inside a block satisfies some criterion (e.g., being less than some threshold). We assume the existence of an auxiliary index, termed the **Count-Index**. The auxiliary index does not contain any data points, but rather maintains the **count** of points in each data block.

We make extensive use of the MINDIST and MAXDIST metrics [19]. Refer to Figure 1 for illustration. The MINDIST (or MAXDIST) between a point, say p, and a block, say b, refers to the minimum (or maximum) possible distance between p and any point in b. Similarly, the MINDIST (or MAXDIST) between two blocks is the minimum (or maximum) possible distance between them. In some scenarios, we process the blocks in a certain order according to their MINDIST from a certain point (or block). An ordering of the blocks based on the MINDIST from a certain point (or block) is termed MINDIST ordering.

Before describing how to estimate the cost of the k-NN operations, we briefly describe the state-of-the-art algorithms for processing the k-NN-Select and k-NN-Join.

Existing k-NN-Select algorithms prune the search space following the branch-and-bound paradigm. [19] applies a depth-first algorithm to read the index blocks in MINDIST order with respect to the query point. Once k points are scanned, the distance between q and the k-farthest point encountered is marked. Refer to Figure 1 for illustration. Assume that $k = 2$. Scanning the blocks starts with Block A (MINDIST = 0). Two points, y and z, are encountered, so the distance between q and z (the farthest) is marked and scanning the blocks continues (Block C then Block B) until the MinDist of a scanned blocks is greater than the distance between q and z. Thus, the overall number of blocks to be scanned is 3.

The above algorithm is suboptimal and cannot be applied for incremental k-NN retrieval. The distance browsing algorithm of [14] achieves optimal performance and can be applied for incremental as well as non-incremental k-NN processing. The main idea of this algorithm is that it can incrementally retrieve the nearest-neighbors to a query point through its `getNextNearest()` method. Two priority queues are maintained: (1) a priority queue for the blocks that have not been scanned yet (blocks-queue for short), and (2) a priority queue for the tuples in the already scanned blocks that have not been returned as nearest-neighbors yet (tuples-queue for short). The entries in the tuples-queue are prioritized based on the distance from the query point, while the entries in the blocks-queue are prioritized based on the MinDist from the query point. Upon an invocation of the `getNextNearest()` method, the top, say t, of the tuples-queue is returned if the distance between t and the query point is less than the MinDist of the top of the blocks-queue. Otherwise, the top of the blocks-queue is scanned and all its tuples are inserted into the tuples-queue (ordered based on the distance from the query point). To illustrate, we apply the distance browsing algorithm to the example in Figure 1. Assume that $k = 2$. Block A is scanned first. Points y and z are inserted into the tuples-queue. The MinDist of Block C is less than the distance of the top of the tuples-queue, and hence Block C is scanned and Point x is inserted into the tuples-queue. Now, Point x is retrieved as the nearest-neighbor followed by Point y. Observe that the algorithm avoids scanning Block B. Thus, the overall number of scanned blocks is 2 that is less than the number of blocks to be scanned if the algorithm in [19] is applied.

In addition to being optimal, the distance browsing algorithm is quite useful when the number of neighbors to be retrieved, i.e., k, is not known in advance. One use case is when a k-NN-Select predicate is combined with a relational predicate within the same query. Consider, for example, a query that retrieves the k-closest restaurants that provide seafood. The distance browsing algorithm gets the nearest restaurant and then examines whether it provides seafood or not. If it is not the case, the algorithm retrieves the next nearest restaurant. This process is repeated until k restaurants satisfying the condition (i.e., provide seafood) are found.

Being the state-of-the-art in k-NN-Select processing, we model the cost of the distance browsing algorithm in this paper. Observe that the cost of the distance browsing algorithm is dominated by the number of blocks that get scanned. Thus, given a k-NN-Select, the goal is to estimate the number of blocks to be scanned without touching the data points. Observe that this goal is challenging because the cost depends on: (1) the value of k, (2) the location of the query point, and (3) the distribution of the data that directly affects the structure of the index blocks. These factors have direct impact on the cost. Refer to Figure 1 for illustration. If the value of k is relatively large, MinDist scanning of the blocks will continue beyond Block C, and thus leading to a larger overall number of scanned blocks. Similarly, if the location of q is different, the MinDist values will change, and thus leading to different block ordering during the MinDist scan, and different overall number of scanned blocks. Also, if the distribution of the data is different, the index blocks will have completely different shapes and locations in space, and this will affect the values of MinDist, and hence will affect the overall number of scanned blocks.

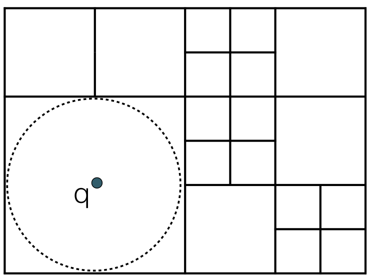
[8, 9, 24] study the problem of estimating the cost of a k-NN-Select operator for uniformly distributed datasets. The authors of [24] further extend their techniques to support non-uniform datasets. The main idea is to estimate the value of D_k (Figure 1), i.e., the smallest radius of a circle centered at the query point and that contains k points. Once the value of D_k is estimated, the number of blocks that overlap with the circle whose center is the query point and whose radius is D_k is determined. This number can be computed by scanning the blocks of the Count-Index in MinDist order from q.

Given a non-uniform dataset, [24] assumes that the points in each block are uniformly distributed and that each block has a constant density. Histograms are maintained to estimate the density of each block in the index. To estimate the cost, [24] applies the following algorithm. The blocks of the Count-Index are scanned in MinDist order from q. Hence, the scanning starts from the block, say b, that is closest (according to MinDist) to q. Observe that if q falls within any block, the MinDist corresponding to that block will be zero, and hence scanning will start from that block. Given the density of Block b, the area of a circle containing k points for that density is computed and then the value of D_k is determined. If the circle is fully contained inside Block b, the search terminates; otherwise, further blocks are examined and the combined density of these blocks is computed. Given the combined density, the area of a circle containing k points is determined. This process is repeated until the computed circle is fully contained within the bounds of the examined blocks. We refer to this algorithm as the density-based algorithm.

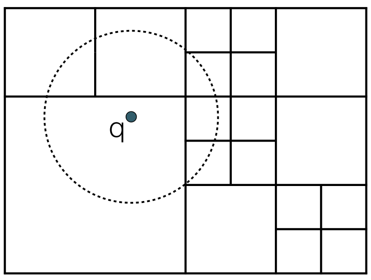
Although the density-based algorithm in [24] achieves good estimation accuracy, it incurs relatively high overhead in many cases. For instance, if the value of k is high or if the density of the blocks around the query point is low, the algorithm will keep extending its search region by examining further blocks until its search region contains k points. In addition, at each iteration of the algorithm, the combined density of the encountered blocks is computed, which can be a costly operation. The process of estimating the cost of a database operator has to be extremely fast. Typically, a database query optimizer keeps a set of catalog information that summarizes the cost estimates. Then, given a query, it performs quick lookups or simple computations to estimate the corresponding cost. With that goal in mind, we propose a new cost estimation technique that incurs no computational overhead at query time, but rather requires $O(1)$ lookups.

Several query processing techniques have been proposed in the literature for processing k-NN-Join operators, e.g., [10, 25, 22]. [22] represents the state-of-the-art technique in k-NN-Join and has proved to achieve better performance than other existing techniques. The key idea that distinguishes [22] from other existing techniques is that in any other technique, each point in a block independently keeps track of its k-nearest-neighbors encountered thus far with no reuse of neighbors of one point as being neighbors of another point in its spatial proximity. In contrast, [22]'s approach identifies a region in space (termed locality) that contains all of the k-nearest-neighbors of all the points in a block. Once the best possible locality is built, each point searches only the locality to find its k-nearest-neighbors. This block-by-block processing methodology results in high performance gains.

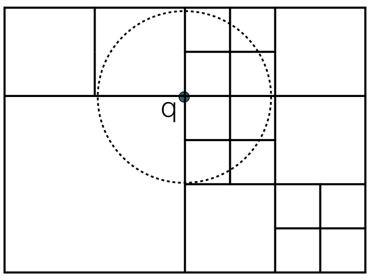
A naive way to estimate the cost of a k-NN-Join operator using the density-based algorithm of [24] is to treat every point from the outer relation as a query point for a k-NN-Select operator and then aggregate the cost across all the points from the outer relation. However, this approach is costly. Furthermore, this approach does not capture the rationale behind the block-by-block processing methodology in k-NN-Join processing as stated above. This calls for efficient cost estimation techniques that can represent the cost of the state-of-the-art techniques in k-NN-Join processing.



(a) Cost = 1



(b) Cost = 6



(c) Cost = 10

Figure 2: Variability of the cost (number of blocks to be scanned) of a query point given its position with respect to the center of the block. Assume that the dashed circle includes exactly k points. The cost tends to increase as the query point gets farther from the center of the block. The maximum cost is at the corners of the block if we assume uniform distribution of the points within the block.

3. K-NN-SELECT COST ESTIMATION

3.1 The Staircase Technique

In this section, we present the Staircase technique; a new technique for estimating the cost (i.e., number of blocks to be scanned) of a k-NN-Select $\sigma_{k,q}(R)$. The main idea of the Staircase technique is to maintain a set of *catalog* information that enables quick estimation of the cost via lookups. Conceptually, the catalog should reflect the cost of a k-NN-Select for every possible query location and for every possible value of k. Given a query point, say q, and the value of k, we can search the catalog and determine the cost. However, maintaining a catalog that covers the domains of these two parameters (k and the location of q) is prohibitively expensive in terms of computation cost and storage requirements. The number of possible locations of q is infinite and the value of k can range from 1 to the size of the underlying table.

One key insight to improve the above approach is to exploit the spatial locality of the k-NN operation to reduce the size of the catalog. We observe that the k-nearest-neighbors of a query point, say q_1, are likely to be among the k-nearest-neighbors of another query point, say q_2, if q_1 and q_2 are within the spatial proximity of each other. In addition, any spatial index structure aims at grouping the points that are within spatial proximity in the same block. This means that the k-nearest-neighbors of the points of the same block have high overlap, and hence the query points that fall within the same block are likely to have similar costs. Given a query point, say q, we can estimate the cost corresponding to q by the cost corresponding to the center of the block in which q is located.

Although the above approach yields good estimation accuracy, it is slightly inaccurate because the cost corresponding to a query point, say q, may vary according to the location of q with respect to the center of the block, say b, in which q is located. For a fixed value of k, the cost corresponding to q is minimum if q is near the center of b and tends to increase as q gets far from the center until it reaches its maximum value in the corners of b. Refer to the example in Figure 2 for illustration of this observation. This observation is particularly true if we assume that within a leaf index block, the points are uniformly distributed. Such assumption is practically reasonable. A typical spatial index tends to split the data points (which can be non-uniformly distributed) until the points are almost balanced across the leaf blocks, and hence points that are within the same block tend to have a uniform distribution within that block.

Applying the above observation, we estimate the cost corresponding to a query point, say q, by combining two values: 1) the cost corresponding to the center of the block, C_{center}, (i.e., the minimum cost), and 2) the cost corresponding to one of the corners,

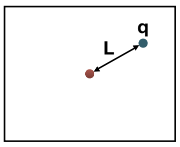


Figure 3: Cost estimation with respect to the center of a block.

C_{corner}, (i.e., the maximum cost). More precisely, the estimated cost can be computed as:

$$Cost = C_{center} + \Delta \cdot \frac{2L}{Diagonal}, \tag{1}$$

where $Diagonal$ is the length of the diagonal of the block, L is the distance between q and the center of the block, and

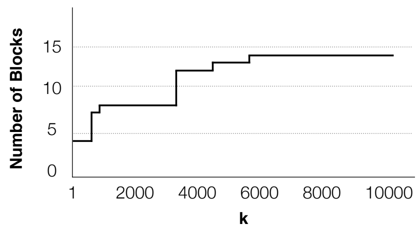
$$\Delta = C_{corner} - C_{center}. \tag{2}$$

Refer to Figure 3 for illustration.

Thus, we do not need to precompute the k-NN cost for every possible query location. Instead, we precompute the cost only for the center and the corners of every block. Although this can reduce the size of the catalog, we still need to precompute the cost for every possible value of k, i.e., from 1 to the size of the table. This can still be prohibitively expensive because it needs to be performed for every block in the index.

We observe that the cost corresponding to any query point tends to be constant for different ranges of values of k. The reason is that the number of points in a block is relatively large, and hence the cost (number of blocks to be scanned) tends to be stable for a range of values of k. To illustrate this idea, consider the example in Figure 1. Assume that Blocks A and B have 1000 points each. Assume further that $k_1 = 500$ tuples in Block A have a distance that is less than the MINDIST betweeb Block B and the query point q. Applying the distance browsing algorithm [14] as explained in Section 2, points in Block A will be inserted in the tuples-queue. The k_1 points in Block A will be retrieved from the tuples-queue before Block B is scanned. Thus, the cost (number of scanned blocks) will equal to 1 for $k \in [1,\ k_1]$. For $k > k_1$, Block B will have to be scanned, and thus the cost will equal to 2. However, the cost will remain equal to 2 for $k \in [k_1 + 1,\ k_2]$, where k_2 equals the number of points in the tuples-queue that have distance less than the MINDIST between Block C and the query point q.

To better illustrate the above observation, we use the OpenStreetMap dataset and build a quadtree index on top (as detailed in Section 4), and then measure the cost corresponding to a random query point. Figure 4 illustrates that the cost is constant for



(a)

$[k_{start}, k_{end}]$	Cost
[1, 520]	3
[521, 675]	7
[676, 3496]	8
[3497, 6799]	12
[4700, 5837]	13
[5838, 10000]	14

(b)

Figure 4: Stability of the cost for different values of k.

large intervals of k.[1] The shape of the graph resembles a staircase diagram (and hence the name *Staircase* for the technique). As the figure demonstrates, the cost is constant for relatively large intervals of k. For instance, when $k \in [1,\ 520]$, the cost is 3 blocks, and when $k \in [521,\ 675]$, the cost is 7 blocks. Observe that this stability increases as the maximum block capacity increases, i.e., the intervals become larger.

We leverage the above stability property to reduce the storage size associated with every block in the index. Instead of blindly computing the cost corresponding to the center (and the corners) of a block for every possible value of k, we determine the values of k at which the cost changes. We store a set of intervals and associate with each interval the corresponding cost. We refer to this information as the *catalog*. The catalog is a set of tuples of the form $([k_{start},\ k_{end}],\ size)$. Refer to Figure 4 for illustration.

3.2 Building the Catalog

The process of building a catalog, be it for the center of a block or for one of the corners, is straightforward. Similarly to the distance browsing algorithm in [14], we maintain two priority queues, a tuples-queue and a blocks-queue. The blocks-queue orders the blocks according to a MINDIST scan. In contrast, the tuples-queue orders the points according to their distance from the query point, say q. We start with the block in which q is located and insert all the block's points into the tuples-queue. At this point, the cost = 1. We keep removing points from the tuples-queue until the MINDIST in the blocks-queue is less than the top of the tuples-queue. The number of points, say k_1, removed so far from the tuples-queue represents the first interval in the catalog, i.e., $([1,\ k_1],\ 1)$. Then, we scan the next block in the blocks-queue, insert all its points into the tuples-queue, and increment the cost. We repeat this process until all the blocks are scanned or a sufficiently large value of k is encountered. Pseudocode of the process of building the catalog of a query point is illustrated in Procedure 1.

For every block in the index, we precompute five catalogs, one for the center and one for each corner. We merge the four catalogs corresponding to the corners into one catalog that stores for each value of k, the maximum cost amongst the four corners. Thus, we store only two catalogs, one for the center (center-catalog, for short), and one that corresponds to the maximum cost at the corners (corners-catalog, for short).

3.3 Cost Estimation

Given a query with a k-NN-Select at Location q, the cost can be estimated as follows. First, we identify the block that encloses q and then search in the center-catalog and the corners-catalog for the

[1]A similar behaviour occurs for any query point, but with different values.

Procedure 1 Building the k-NN-Select-Cost Catalog.

```
Terms: $q$: The query point to which we need to build the catalog.
    $MAX\_K$: The maximum possible/maintained value of $k$.
 1: $tupleQ \leftarrow \emptyset$; $blockQ \leftarrow$ MINDIST scan w.r.t. $q$
 2: $cost \leftarrow 0$; $currentK \leftarrow 1$; $catalog \leftarrow \emptyset$
 3: while ($currentK < MAX\_K$) do
 4:   $currentBlock \leftarrow blockQ.next()$
 5:   $cost++$
 6:   $tupleQ.insert(currentBlock.allPoints)$ ordered according to the distance from q
 7:   $startK \leftarrow currentK$
 8:   while ($tupleQ.top.distance \leq blockQ.top.$MINDIST ) do
 9:     $tupleQ.removeTop()$
10:     $currentK++$
11:   end while
12:   $catalog.add([startK,\ currentK],\ cost)$
13: end while
14: return $catalog$
```

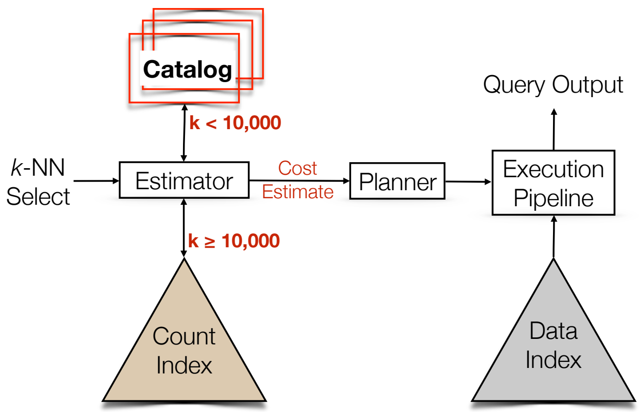


Figure 5: Cost estimation for a k-NN-Select.

intervals to which the value of k belongs. Observe that the above process for building a catalog yields a sorted list of ranges of values of k, and hence binary search can be applied to find the enclosing interval and the corresponding cost in logarithmic time w.r.t. the number of intervals. Then, the cost is estimated using Equations 1 and 2.

Because the Staircase technique relies on precomputing the estimates, the auxiliary index that contains the statistics, e.g., counts and cost estimates, has to be a space-partitioning index, e.g., quadtree or grid, so that the query point always falls inside a block. Observe that the structure of the auxiliary index can be independent of the index that contains the actual data points, i.e., the data-index. If the data-index is a space partitioning index, then the auxiliary index can have the same exact structure as the data-index. If the data-index is a data-partitioning index, e.g., R-Tree, then the structure of the auxiliary index will be different. In either case, the query point will never be outside a block in the auxiliary index, and hence we will always be able to estimate the cost

Because the number of blocks in the index can be large, the storage overhead of the catalogs can be significant. We hence limit the maximum value of k that a catalog supports to a practically large constant, e.g., $k = 10,000$. This would result in compact catalogs that can be practically maintained for each index block. In the case when a k-NN-Select query has a k value that is greater than $10,000$, we can estimate its cost by applying the algorithm

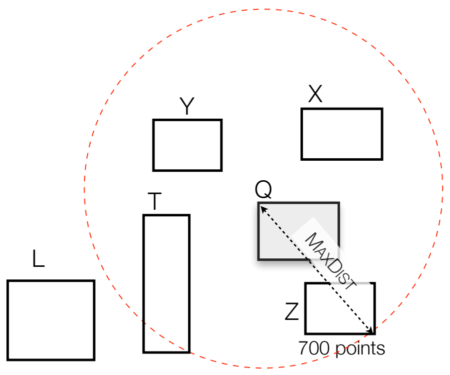


Figure 6: Building the locality of a block. The gray block Q is a block from the outer relation; other blocks are from the inner relation.

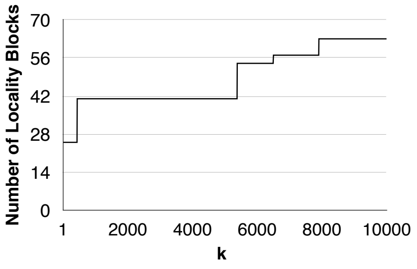
in [24] using the Count-Index. Figure 5 illustrates the typical flow of a k-NN-Select query. Queries with $k > 10,000$ (that do not arise frequently in practice) are directed to the Count-Index. All other queries ($k \leq 10,000$) are served through the catalogs. In Section 5, we show that for a real dataset of 0.1 Billion points, the overhead to store all the catalogs is less than 4 MBs.

4. K-NN-JOIN COST ESTIMATION

As highlighted in Section 2, the state-of-the-art technique [22] in k-NN-Join processing applies a block-by-block mechanism in which, for each block from the outer relation, the locality blocks are determined from the inner relation. The locality blocks of a block, say b_o, from the outer relation represent the minimal set of blocks in which the k-nearest-neighbors of any point $\in$ bo exist. Thus, each point from the outer relation searches only the locality of its enclosing block to find its k-nearest-neighbors.

Before estimating the cost of k-NN-Join, we briefly explain how the locality of a block is computed. Given a block from the outer relation, say b_o, the corresponding locality blocks in the inner relation are determined as follows. Blocks of the inner relation are scanned in MINDIST order from b_o. The sum of the count of points in the encountered blocks is maintained. Once that sum reaches k, the highest MAXDIST, say M, of an encountered block is marked and scanning of the blocks continues until a block of MINDIST greater than M is encountered. The encountered blocks represent the locality of b_o. For example, consider the process of finding the locality of Block Q in Figure 6 where $k = 10$. Blocks are scanned in MINDIST order from Block Q. This means that scanning starts with Block Z. Assume that Block Z contains 700 points. Now, the sum of the count of points in the encountered blocks (in this case, only Block Z) exceeds k. The MAXDIST between Block Q and Block Z is marked, and scanning the blocks continues (Blocks X, Y, and T, respectively) until Block L is encountered. At this point, scanning is terminated because Block L has MINDIST from Block Q that is greater than the marked MAXDIST (between Blocks Q and Z). Hence, the number of blocks in the locality of Block Q is 4.

A naive way to estimate the cost (i.e., the total number of scanned blocks) of a k-NN-Join is to compute the size of the locality blocks for each block in the outer relation and sum these sizes. However, this can be expensive because the number of blocks in the outer relation can be arbitrarily large. In the rest of this section, we present three different techniques that address this problem.



(a)

[kstart, kend]	Locality Size
[1, 313]	25
[314, 5380]	41
[5381, 6537]	54
[6538, 9368]	57
[9369, 10882]	63

(b)

Figure 7: Stability in the size of the locality for different values of k.

4.1 The Block-Sample Technique

Instead of computing the locality for each block of the outer relation, we pick a random sample of these blocks, compute the locality size of each block in the sample, and then aggregate the total size and scale it to the total number of blocks in the outer relation. We refer to this technique as the Block-Sample technique.

Given a set of n_o blocks from the outer relation, we pick a random sample of size s. If the aggregate locality size of the s blocks is agg, then we estimate the overall join cost as $\frac{agg \times n_o}{s}$. The sample blocks are chosen to be *spatially distributed* across the space in which the blocks of the outer relation reside. To get such sample of blocks, we do either a depth-first or breadth-first index traversal for the blocks of the outer relation and skip blocks every $\frac{n_o}{s}$.

Although the above technique can result in high accuracy when the sample size increases, it incurs computational overhead upon receiving a k-NN-Join query. As mentioned earlier in Section 2, a typical query optimizer requires fast estimation of the cost, possibly through quick catalog-lookups. With this goal in mind, we introduce next a catalog-based approach.

4.2 The Catalog-Merge Technique

The main idea of the Catalog-Merge technique is to precompute the size of the locality of each block in the outer relation and store it in a catalog. Given a k-NN-Join query, we can simply aggregate the precomputed values in the catalogs of the blocks of the outer relation. However, the size of the locality depends on the value of k, so we need to precompute it for every possible value of k, which can be prohibitively expensive.

Similarly to the case of k-NN-Select, we observe that the size of the locality of a given block tends to be constant (stable) for relatively large intervals of k. To illustrate, consider the example in Figure 6. Assume that Block Z has 700 points. If k has any value between 1 and 700, exactly the same set of blocks will represent the locality of Block Z, i.e., the size of the locality will be the same. To better illustrate this observation, we use the OpenStreetMap dataset and build a quadtree index on top (as detailed in Section 4), and then measure the locality size of a randomly selected block.[2] Figure 7 illustrates that the size of the locality is stable for large intervals of k.

To build the locality-catalog of a block, we identify the inflection points in the range of values of k at which the locality size changes, e.g., $k = 313, 5380, \ldots, 9368$ in Figure 7. This can be performed using binary search within the range of values of k. In particular, we start with $k = 1$ and compute the locality size, say S. Then, we perform a binary search for the smallest value of k at which the

[2]A similar behaviour occurs for any block, but with different values.

locality size would be greater than S, i.e., the inflection point, say k_i. At this moment, we identify the first range of values as $[1,\ k_i - 1]$. Afterwards, we perform another binary search starting from $k = k_i$ to get another range of values of k. This process is repeated until no inflection points are found, i.e., when the maximum value of k is reached.

A more efficient approach is to build the locality-catalog *incrementally* through two rounds of MinDist scan of the Count-Index. These MinDist scan rounds can be achieved using *two* priority queues in which the blocks of the Count-Index are ordered according to their MinDist from the block we need to build the catalog for. The two MinDist scan rounds are interleaved. One scan explores the blocks that should contain at least C points. We refer to this scan as Count-Scan. The other scan explores the blocks that have MinDist $\leq$ the highest MaxDist value of the explored blocks so far from Count-Scan. We refer to this queue as Max-Scan. We maintain a counter, say C. Whenever a block from Count-Scan is retrieved, its MaxDist, say M, is marked and the value of C is incremented by the number of points in the retrieved block. Then, blocks from Max-Scan are scanned until the MinDist is greater than the highest value encountered for M. At this point, a new entry is created in the catalog by aggregating the number of blocks retrieved from Max-Scan thus far. This process is repeated until C reaches the maximum value of k or all the blocks of the inner relation are consumed by Count-Scan.. Pseudocode for the process is given in Procedure 2. Refer to Figure 6 for illustration, where we compute the catalog of Block Q. We start with $C = 1$. In Count-Scan, we explore Block Z, update C to be 700, and mark the highest MaxDist encountered. Then, in Max-Scan, we explore Blocks X, Y, and T because their MinDist is less than the largest MaxDist encountered. At this moment, we create a catalog entry $([1, 700],\ 4)$ that represents the start and end values of C with the cost of four blocks (namely, Z, X, Y and T). Afterwards, we continue Count-Scan to explore Block X. Assuming that Block X has 500 points, we update C to be $700 + 500 = 1200$ and also mark the MaxDist between Blocks X and Q. Then, in Max-Scan, we explore Block L because its MinDist is less than the highest MaxDist marked thus far. Now, the cost is incremented by 1 due to Block L. We create a new catalog entry $([701, 1200],\ 5)$.

Observe that the above approach is cheap because it relies only on counting (using the Count-Index) with no scan of the data. Assume that for a given block from the outer relation, the number of blocks in its locality is L. The above approach visits each of the L blocks at most twice, i.e., the running time is $O(L)$ per block.

Note that, by definition, the *locality* conservatively includes all the blocks needed for the k-NN search (see [22] for details), i.e., the locality contains the k-NN for every point in the outer block, say Q. Although it is true that for some $k_1 > k$ all the nearest-neighbors of some points in Q may exist in already scanned blocks (by Count-Scan), there will be some points, e.g., near the corners of Q, that might have some of their k-NN in unscanned blocks. Hence, in our approach, we jump into new ranges of k (and new corresponding cost) whenever a block is retrieved through Count-Scan.

Observe that if a block retrieved from Count-Scan has MaxDist that is less than or equal to the highest MaxDist encountered thus far, it will not lead to any scan in Max-Scan, and hence will lead to a repeated cost in the next entry of the catalog. For instance, in Figure 6, if the MaxDist between Blocks Q and Z is greater than the MaxDist between Block Q and Block X (the next in Count-Scan), then the next new entry in the catalog will be $([701, \ldots],\ 4)$, i.e., will have the same cost. To get rid of these redundant entries in the catalog, we continue Count-Scan until the value of the highest encountered MaxDist changes.

Procedure 2 Building the locality-catalog of a block.

```
Terms: $Q$: The block to which we need to build the catalog.
    $MAX\_K$: The maximum possible/maintained value of $k$.
 1: // Initializations:
 2: $CountScan \leftarrow$ MinDist Scan from $Q$
 3: $cBlock \leftarrow CountScan.next()$
 4: $MaxScan \leftarrow$ MinDist Scan from $Q$
 5: $mBlock \leftarrow MaxScan.next()$
 6: $C \leftarrow 1$; $aggCost \leftarrow 0$;
 7: $Catalog \leftarrow \emptyset$; $highestMaxDist \leftarrow 0$
 8: while ($C < MAX\_K$) do
 9:   $startK \leftarrow C$
10:   while ($cBlock$.MaxDist $\leq highestMaxDist$) do
11:     $C+ = cBlock.count$
12:     $cBlock \leftarrow CountScan.next()$
13:   end while
14:   $highestMaxDist \leftarrow cBlock$.MaxDist from $Q$
15:   $endK \leftarrow C$
16:   while ($mBlock$.MinDist $\leq highestMaxDist$) do
17:     $aggcost + +$
18:     $mBlock \leftarrow MaxScan.next()$
19:   end while
20:   $Catalog.add([startK,\ endK],\ aggCost)$
21: end while
22: return $Catalog$
```

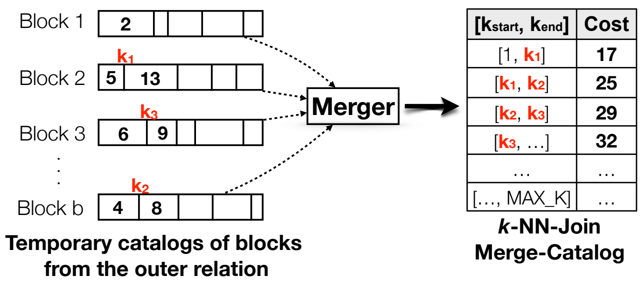


Figure 8: Flow of the Catalog-Merge process.

4.2.1 *Preprocessing*

For each block in the outer relation of the k-NN-Join, we compute a temporary catalog that is similar to the one in Figure 7(b). If the number of blocks in the outer relation is n_o, then this process requires $O(n_o \cdot L)$, where L is the average size of the locality of a block. This can be costly if n_o is large. To solve this problem, we take a spatially distributed random sample of the blocks of the outer relation. We compute a temporary catalog only for the sample blocks and not for each block in the outer relation. Afterwards, we merge all the temporary catalogs, and produce a single catalog that contains the *aggregate* cost of all the temporary catalogs. Each entry in the final catalog has the form $([k_{start},\ k_{end}],\ size)$, where $size$ is the estimated join cost when $k_{start} \leq k \leq k_{end}$.

Because each temporary catalog is sorted with respect to the ranges of values of k, we apply a plane sweep over the ranges of values of k and aggregate the cost. To illustrate, consider the example in Figure 8. k_1 is the smallest value of k in the catalog entries. This means that the aggregate cost for the interval $[1,\ k_1]$ is $2+5+6+4 = 17$. k_2 is the next smallest value of k, and hence another interval $[k_1,\ k_2]$ with aggregate cost = $17 - 5 + 13 = 25$ is created in the output catalog. Similarly, for interval $[k_2, k_3]$, the aggregate cost = $25 - 4 + 8 = 29$ and for interval $[k_3,\ \cdots]$, the aggregate cost = $29 - 6 + 9 = 32$. A min-heap is used to efficiently determine the next smallest value across all the temporary catalogs in the plane sweep process.

To reduce the size of the catalog, we limit the maintained values of k to some practically large constant, e.g., 10,000. In Section 5, we show that for a real dataset of 0.1 Billion points, the size of the catalog is about 1 MB.

4.2.2 Cost Estimation

Observe that the resulting k-NN-Join catalog is sorted w.r.t. the values of k. Given a k-NN-Join query, we can lookup the estimate cost using a binary search to find the catalog entry corresponding to the given k value.

Although the process of building the catalog is performed once, it can be costly if the number of tables in the database schema is large, say n. The k-NN-Join catalog information is required for every possible pair of tables in the database schema, and hence $2 \times \binom{n}{2}$ catalogs need to be built (because the k-NN-Join is asymmetric). Although sampling can speed up the merging process of the temporary catalogs, it is still expensive to compute $2 \times \binom{n}{2}$, i.e., a quadratic number of catalogs across the database tables. To address this issue, we introduce our third cost estimation technique that requires only a linear number of catalogs.

4.3 The Virtual-Grid Technique

Similarly to the Catalog-Merge technique, in the Virtual-Grid technique, we maintain a set of catalog information that is built only once before executing any queries. The key idea is to estimate the cost corresponding to a dataset, say D, when D is the inner relation of a k-NN-Join. Given the n relations in the database schema, where each can potentially be an outer relation in a k-NN-Join with D, instead of computing n catalogs corresponding to D, we compute only one catalog that corresponds to the join cost between a virtual index and D.

4.3.1 Preprocessing

Refer to Figure 9 for illustration. Given the index of a dataset (e.g., the red quadtree decomposition in the figure), we assume the existence of a virtual grid that covers the whole space.[3] For each block (grid cell) in the virtual-grid, we compute a catalog that is similar to the one in Figure 7(b) with the difference that the locality is computed with respect to the given index. We associate all these virtual-grid catalogs with the given index (e.g., the quadtree). We repeat this process for each relation in the database schema, i.e., associate with every index a virtual-grid-cost. Observe that unlike the Catalog-Merge approach, this requires linear storage (and preprocessing time) overhead.

4.3.2 Cost Estimation

Given a k-NN-Join query, we retrieve the virtual-grid corresponding to the inner relation. Then, we estimate the cost by scaling the cost corresponding to the part of the virtual-grid that overlaps with the outer relation. In particular, for each grid cell, say C, in the virtual-grid, we retrieve the locality size, say L, stored in C's catalog. Then, we select the blocks in the outer relation that overlap with C. This can be performed using a range query on the outer relation. For each of the overlapping blocks, say O, in the outer relation, we multiply L by the ratio between the diagonal length of Block O and the diagonal length of Block C. We sum these products across all the cells of the virtual-grid. The overall sum represents the join cost estimate.

[3]This can be achieved for real datasets where the bounds of the earth are fixed.

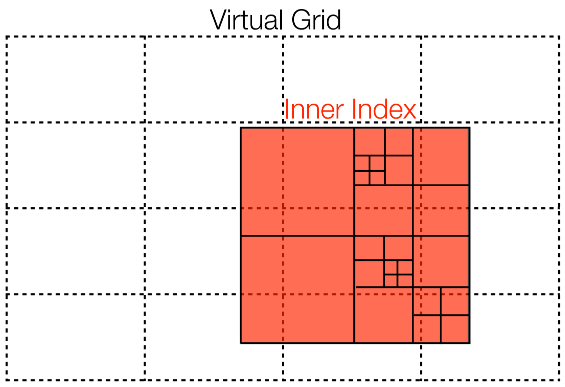


Figure 9: The Virtual-Grid technique for k-NN-Join cost estimation.

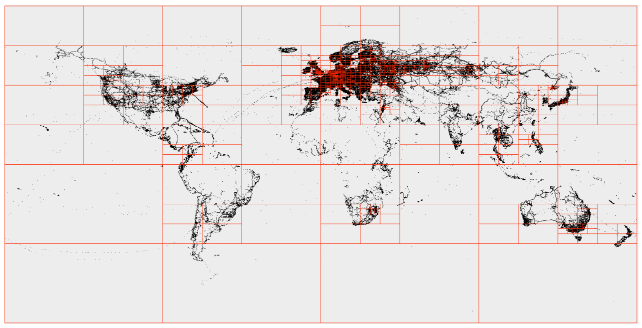


Figure 10: A sample of OpenStreetMap GPS data and the corresponding region-quadtree decomposition overlaid on top.

Assuming that the number of blocks in the outer relation is n_o, the estimation process is $O(n_o)$. The reason is that eventually, all the blocks of the outer relation get selected (through the range query performed at each grid cell). In other words, regardless of the grid size, all the blocks will be selected and the corresponding products have to be aggregated. In Section 5, we study the estimation time for different grid sizes while fixing the size of the outer relation and demonstrate that the estimation time is almost constant for different grid sizes.

5. EXPERIMENTS

In this section, we evaluate the performance of the proposed estimation techniques. We realize a testbed in which we implement the state-of-the-art techniques for k-NN-Select estimation [24] as well as our proposed estimation techniques. To have a ground truth for the actual cost of the k-NN operators, we implement the Distance Browsing algorithm for k-NN-Select as well as the locality based k-NN-Join. Our implementation is based on a region-quadtree index [21], where each node in the quadtree represents a region of space that is recursively decomposed into four equal quadrants, or subquadrants, with each leaf node containing points that correspond to a specific subregion. The maximum block capacity in the quadtrees used in our experiments is set to 10,000 points. All implementations are in Java. Experiments are conducted on a machine running Mac OS X on Intel Core i7 CPU at 2.3 GHz and 8 GB of main memory.

We use a real spatial dataset from OpenStreetMap [1]. The number of data points in the dataset is 0.1 Billion points. Figure 10 displays a sample of the data that we plot through a visualizer that

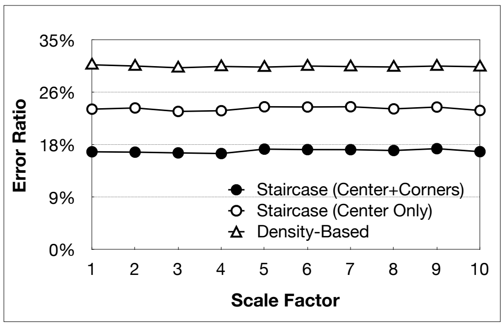


Figure 11: k-NN-Select estimation accuracy.

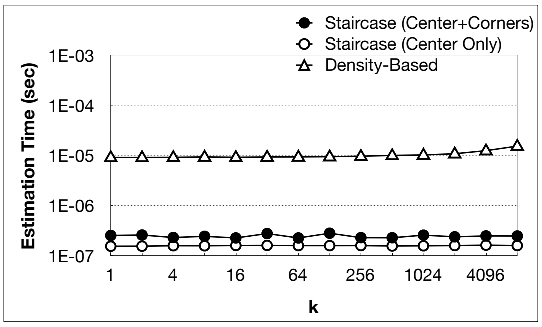


Figure 12: k-NN-Select estimation time.

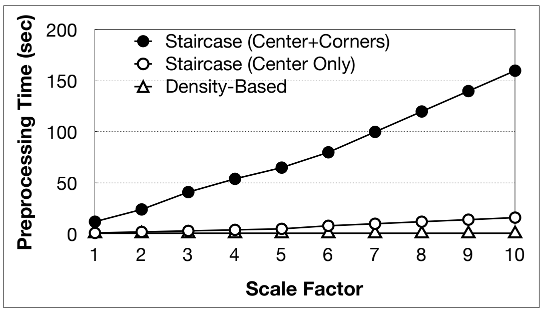


Figure 13: Preprocessing time of the k-NN-Select estimation techniques.

we have built as part of our testbed. The figure also displays a region-quadtree decomposition that is built on top of the data.

To test the performance of our techniques at different data scales, we insert portions of the dataset into the index at multiple ratios. For instance, for scale = 1, we insert 10 Million points, for scale = 2, we insert 20 Million points, and so on until scale = 10 in which all the 0.1 Billion points are inserted. Our performance metrics are the estimation accuracy (i.e., error ratio), the estimation time, the preprocessing time, and the storage overhead. We limit the maximum maintained value of k in all the catalogs to 10,000.

5.1 KNN-Select Cost Estimation

In this section, we present the performance of the Staircase technique in estimating the cost of a k-NN-Select and compare it with the density-based technique of [24]. We evaluate two variants of the Staircase technique, 1) Center-Only, where the cost corresponding to a query point, say q, is estimated as the cost corresponding to the center in which q is located, and 2) Center+Corners, where the cost is estimated using Equations 1 and 2.

5.1.1 Estimation Accuracy

In this experiment, we measure the average error ratio in estimating the cost of 100,000 queries that are chosen at random. For each query, we compute the actual cost, compare it with the estimated cost, and measure the error ratio. We compute the average error ratio of all the queries.

Figure 11 illustrates that the Staircase technique achieves a smaller error ratio than that of the density-based technique. The error ratio reaches less than 20% when the cost is estimated using the Center+Corners variant.

5.1.2 Estimation Time

In this experiment, we measure the time each estimation technique requires to estimate the cost of a query. Figure 12 illustrates that the Staircase technique is almost two orders of magnitudes faster than the density-based technique. Observe that the Center+Corners variant of the Staircase technique is slightly slower than the Center-Only variant because the former requires two catalog lookups, one from the center-catalog and the other from the corners-catalog. Also, observe that the estimation time of the density-based technique increases as the value of k increases. The reason is that the density-based technique keeps scanning the index blocks until the encountered blocks are estimated to contain k points. In contrast, the estimation time of the Staircase technique is constant regardless of the value of k because the Staircase technique relies on just a single catalog lookup (two lookups in case of the Center+Corners variant).

5.1.3 Storage Overhead and Preprocessing Time

In this experiment, we measure the storage requirement and preprocessing time of each estimation technique. Observe that the density-based technique has no preprocessing time requirements because it precomputes no catalogs.

Figure 13 illustrates that the Staircase technique incurs relatively high preprocessing overhead to precompute the catalogs of all the index blocks. Observe that as the scale factor increases, the preprocessing time increases because more blocks will need to be processed. Also, observe that the Center-Only variant incurs less preprocessing overhead than the Center+Corners variant because the former computes only one catalog per block while the latter computes five catalogs and merges four of them. Notice that this preprocessing phase is an offline process that does not affect the performance of the online cost estimation process.

Figure 14 illustrates that density-based technique consumes little storage overhead, basically, due to the density values maintained at each block in the index. In contrast, the Staircase technique has higher storage overhead due to the maintained catalogs. Observe that as the scale factor increases, the storage overhead increases because more blocks will be present in the index and each of them will have a separate catalog. However, the storage requirements of the Staircase technique are less than 4 MBs even for the largest scale factor. Also, observe that the Center-Only variant of the Staircase technique incurs less storage overhead than the Center+Corners variant because the former maintains only one catalog per block while the latter maintains two catalogs.

5.2 K-NN-Join Cost Estimation

In this section, we study the performance of the proposed techniques for estimating the k-NN-Join cost, namely the Block-Sample, Catalog-Merge, and Virtual-Grid techniques.

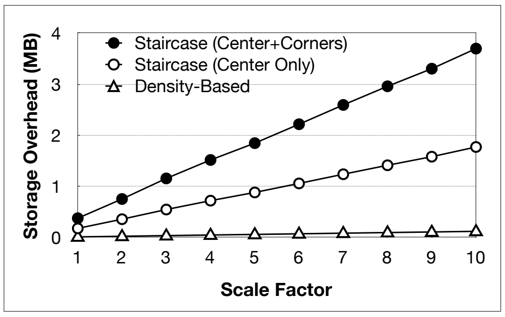


Figure 14: Storage requirements of the k-NN-Select estimation techniques.

5.2.1 Estimation Accuracy

In this experiment, we estimate the cost of a k-NN-Join between two indexes of 0.1 Billion points each for a random value of k, compare it with the actual cost, and then calculate the error ratio. We repeat this process for various sampling sizes for both the Block-Sample and Catalog-Merge techniques, and for various grid sizes for the Virtual-Grid technique. Figure 15 illustrates that the Block-Sample and Catalog-Merge techniques can reach an error ratio that is less than 5% for a sample size > 400. Figure 16 illustrates that the Virtual-Grid technique achieves less than 20% error ratio.

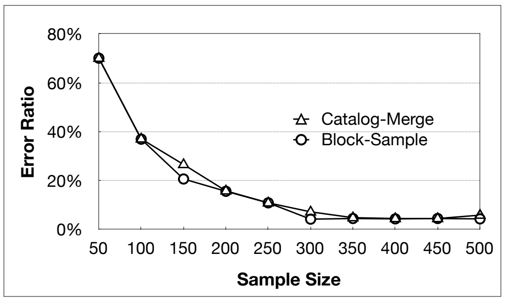


Figure 15: k-NN-Join estimation accuracy.

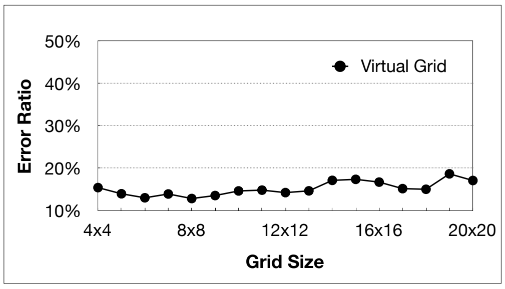


Figure 16: k-NN-Join estimation accuracy.

5.2.2 Estimation Time

In this experiment, we measure the time required to estimate the cost of a k-NN-Join between two indexes of 0.1 Billion points each. Figure 17 gives the performance for different values of k. The number of samples used in the Catalog-Merge and Block-Sample techniques is fixed to 1000. The grid size used in the Virtual-Grid technique is 10×10. As the figure demonstrates, the Catalog-Merge technique is more than four orders of magnitude faster than the Block-Sample and Virtual-Grid techniques. The reason for this variance in performance is that the Catalog-Merge technique maintains one catalog for every pair of relations (indexes) in which the estimate cost is maintained; the cost is directly retrieved from the catalog via one lookup. In contrast, the Block-Sample technique computes the locality for a sample of blocks, which is costly. Also, the Virtual-Grid technique aggregates the cost across each of the grid cells after computing the overlap with the outer relation, which is costly as well.

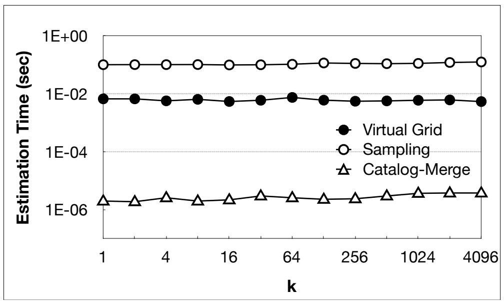


Figure 17: k-NN-Join estimation time.

Figure 18 gives the performance of the Block-Sample and Catalog-Merge techniques for different sample sizes. Observe that the estimation time of the Block-Sample technique increases as the sample size increases.[4] In contrast, the estimation time of the Catalog-Merge technique is constant irrespective of the sample size because estimation is performed through one lookup through a pre-computed catalog, i.e., the sample size only affects the preprocessing time as we show next.

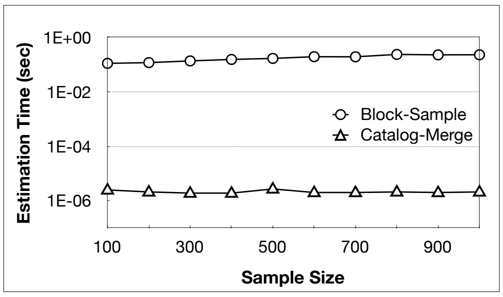


Figure 18: k-NN-Join estimation time.

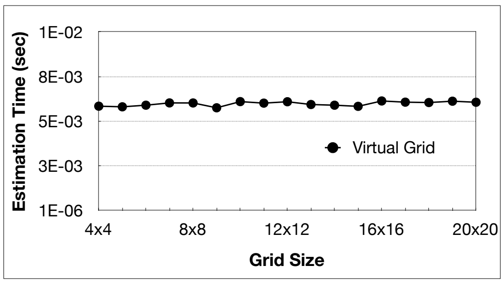


Figure 19: k-NN-Join estimation time.

[4]The slope of the curve is low due to the use of a log-scale.

Figure 19 gives the performance of the Virtual-Grid technique for different grid sizes. As the figure demonstrates, the estimation time is almost constant regardless of the grid size. As highlighted in Section 4, the reason is that the time required for estimation depends on the number of blocks in the outer relation, not on the number of cells in the grid. For each grid cell, the overlapping blocks from the outer relation have to be retrieved regardless of the size of the grid.

5.2.3 Storage Overhead and Preprocessing Time

In this experiment, we measure the storage and preprocessing time requirements for maintaining a set of catalogs for the estimation of k-NN-Join queries between 10 indexes that we create. We test the performance at different scale factors, i.e., create 10 different indexes for each scale factor. For instance, if the scale factor is 5, this means that we create 10 indexes and insert 50 Million points into each of them.

In Figure 20, we fix the grid size in the Virtual-Grid technique to 10×10 and the sample size for the Catalog-Merge technique to 1000. As the figure demonstrates, the Virtual-Grid technique requires almost an order of magnitude less storage than the Catalog-Merge techniques. The reason is that the Catalog-Merge technique maintains a catalog for every pair of indexes, i.e, $2 \times \binom{10}{2} = 90$ catalogs. In contrast, the Virtual-Grid technique maintains a catalog for every index, i.e., only 10 catalogs. Figure 21 demonstrates that the Virtual-Grid technique requires a constant amount of preprocessing time (about two seconds) regardless of the scale factor. The reason is that the preprocessing time depends on the number of grid cells; for each grid cell, a catalog is computed.

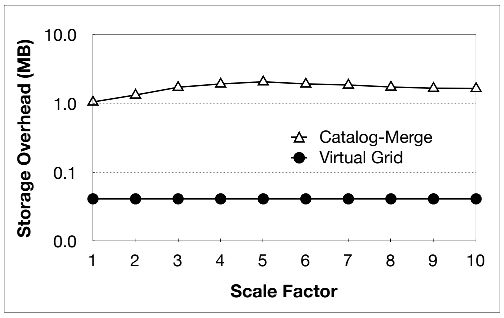


Figure 20: Storage requirements of the k-NN-Join estimation techniques.

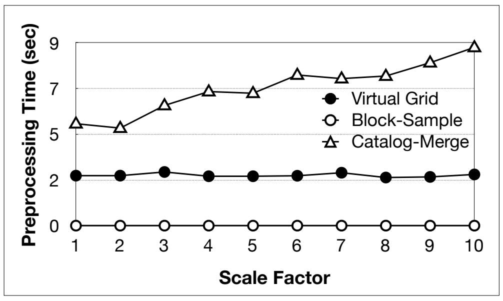
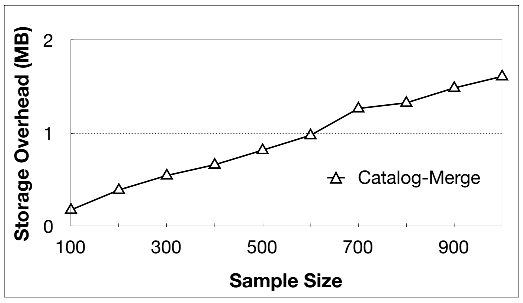
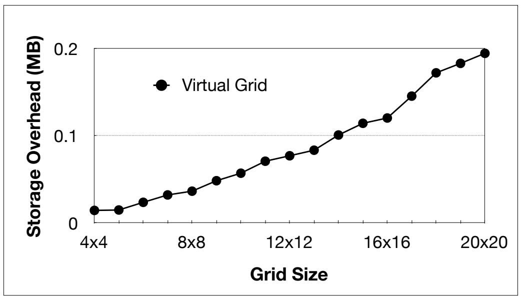


Figure 21: Preprocessing time of the k-NN-Join estimation techniques.

In Figure 23, we fix the scale factor to 10. As Figs. 22(a) and 23(a) demonstrate, the Catalog-Merge technique requires more storage and preprocessing time as the sample size increases. The reason is that, as the sample size increases, more temporary catalogs get created during the process of merging the catalogs, which are likely to result in more entries in the final merged catalog. Similarly, Figs. 22(b) and 23(b) demonstrate that the Virtual-Grid technique requires more storage and preprocessing time as the grid size increases because it maintains a catalog for every grid cell.

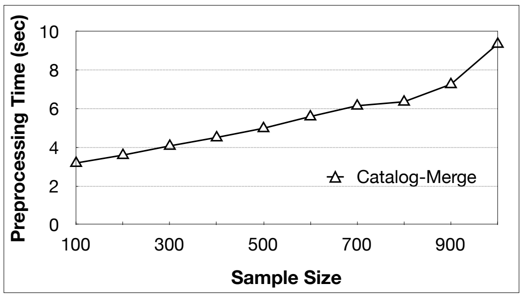


(a)

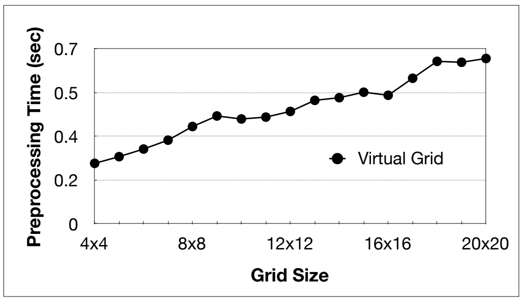


(b)

Figure 22: Storage requirements of the k-NN-Join estimation techniques.



(a)



(b)

Figure 23: Preprocessing time of the k-NN-Join estimation techniques.

		Estimation Time	Estimation Accuracy	Storage Overhead	Preprocessing Time
k-NN-**Select** Cost Estimation	**Density-Based**	Medium	Medium	None	None
	Staircase (Center-Only)	Low	Medium	Low	Medium
	Staircase (Center+Corners)	Low	High	Low	High
k-NN-**Join** Cost Estimation	**Block-Sample**	High	High	None	None
	Catalog-Merge	Low	High	Medium	Medium
	Virtual-Grid	Medium	Medium	Low	Low

Figure 24: Summary of the pros and cons of each estimation technique.

6. CONCLUDING REMARKS

In this paper, we study the problem of estimating the cost of the k-NN-Select and k-NN-Join operators. We present various estimation techniques; Figure 24 summarizes the tradeoffs each technique offers. Performance evaluation using real spatial datasets from OpenStreetMap demonstrates that: 1) the Staircase technique for k-NN-Select cost estimation is faster than the state-of-the-art technique [24] by more than two orders of magnitude and has better estimation accuracy; 2) the Catalog-Merge and Virtual-Grid techniques for k-NN-Join cost estimation achieve less than 5% and 20% error ratio, respectively, while keeping the estimation time below one microsecond and one millisecond, respectively; and 3) the Virtual-Grid technique reduces the storage required to maintain the catalogs by an order of magnitude compared to the Catalog-Merge technique.

7. REFERENCES

[1] OpenStreetMap bulk gps point data. `http://blog.osmfoundation.org/2012/04/01/bulk-gps-point-data/`.

[2] S. Acharya, V. Poosala, and S. Ramaswamy. Selectivity estimation in spatial databases. In *SIGMOD Conference*, pages 13–24, 1999.

[3] N. An, Z.-Y. Yang, and A. Sivasubramaniam. Selectivity estimation for spatial joins. In *ICDE*, pages 368–375, 2001.

[4] W. G. Aref and H. Samet. Estimating selectivity factors of spatial operations. In *FMLDO*, pages 31–43, 1993.

[5] W. G. Aref and H. Samet. A cost model for query optimization using R-Trees. In *ACM-GIS*, pages 60–67, 1994.

[6] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD Conference*, pages 322–331, 1990.

[7] A. Belussi and C. Faloutsos. Estimating the selectivity of spatial queries using the 'correlation' fractal dimension. In *VLDB*, pages 299–310, 1995.

[8] S. Berchtold, C. Böhm, D. A. Keim, and H.-P. Kriegel. A cost model for nearest neighbor search in high-dimensional data space. In *PODS*, pages 78–86, 1997.

[9] C. Böhm. A cost model for query processing in high dimensional data spaces. *ACM Trans. Database Syst.*, 25(2):129–178, 2000.

[10] C. Böhm and F. Krebs. The k-nearest neighbour join: Turbo charging the kdd process. *Knowl. Inf. Syst.*, 6(6):728–749, 2004.

[11] S. Chandrasekaran and M. J. Franklin. Streaming queries over streaming data. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*, pages 203–214, 2002.

[12] M. Y. Eltabakh, R. Eltarras, and W. G. Aref. Space-partitioning trees in postgresql: Realization and performance. In *ICDE*, page 100, 2006.

[13] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD Conference*, pages 47–57, 1984.

[14] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24(2):265–318, 1999.

[15] Y.-W. Huang, N. Jing, and E. A. Rundensteiner. A cost model for estimating the performance of spatial joins using R-trees. In *SSDBM*, pages 30–38, 1997.

[16] H.-P. Kriegel, P. Kunath, and M. Renz. R*-tree. In *Encyclopedia of GIS*, pages 987–992. 2008.

[17] N. Mamoulis and D. Papadias. Selectivity estimation of complex spatial queries. In *SSTD*, pages 155–174, 2001.

[18] V. Poosala, Y. E. Ioannidis, P. J. Haas, and E. J. Shekita. Improved histograms for selectivity estimation of range predicates. In *SIGMOD Conference*, pages 294–305, 1996.

[19] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD Conference*, pages 71–79, 1995.

[20] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann Publishers Inc., 2006.

[21] H. Samet and R. E. Webber. Storing a collection of polygons using quadtrees. *ACM Trans. Graph.*, 4(3):182–222, 1985.

[22] J. Sankaranarayanan, H. Samet, and A. Varshney. A fast all nearest neighbor algorithm for applications involving large point-clouds. *Computers & Graphics*, 31(2):157–174, 2007.

[23] C. Sun, D. Agrawal, and A. El Abbadi. Selectivity estimation for spatial joins with geometric selections. In *EDBT*, pages 609–626, 2002.

[24] Y. Tao, J. Zhang, D. Papadias, and N. Mamoulis. An efficient cost model for optimization of nearest neighbor search in low and medium dimensional spaces. *IEEE Trans. Knowl. Data Eng.*, 16(10):1169–1184, 2004.

[25] C. Xia, H. Lu, B. C. Ooi, and J. Hu. Gorder: An efficient method for KNN join processing. In *VLDB*, pages 756–767, 2004.

Reconstruction Privacy: Enabling Statistical Learning

Ke Wang
Simon Fraser University
Singapore Management University
wangk@cs.sfu.ca

Chao Han
Simon Fraser University
hanchaoh@cs.sfu.ca

Ada Waichee Fu
Chinese University of Hong Kong
adafu@cse.cuhk.edu.hk

Raymond Chi Wing Wong
Hong Kong University of Science and Technology
raywong@cse.ust.hk

Philip S. Yu
University of Illinois at Chicago
Institute for Data Science
Tsinghua University (Beijing)
psyu@cs.uic.edu

ABSTRACT

Non-independent reasoning (NIR) allows the information about one record in the data to be learnt from the information of other records in the data. Most posterior/prior based privacy criteria consider NIR as a privacy violation and require to smooth the distribution of published data to avoid sensitive NIR. The drawback of this approach is that it limits the utility of learning statistical relationships. The differential privacy criterion considers NIR as a non-privacy violation, therefore, enables learning statistical relationships, but at the cost of potential disclosures through NIR. A question is whether it is possible to (1) allow learning statistical relationships, yet (2) prevent sensitive NIR about an individual. We present a data perturbation and sampling method to achieve both (1) and (2). The enabling mechanism is a new privacy criterion that distinguishes the two types of NIR in (1) and (2) with the help of the law of large numbers. In particular, the record sampling effectively prevents the sensitive disclosure in (2) while having less effect on the statistical learning in (1).

Categories and Subject Descriptors

H.2.7 [**Database Management**]: Database Administration—*Security, integrity, and protection*; H.2.8 [**Database Applications**]: Data Mining

General Terms

Algorithm, Data Privacy, Theory

Keywords

Data Privacy, Differential Privacy

1. INTRODUCTION

1.1 Motivation

Many privacy definitions/criteria have been proposed in the literature and many ways exist to categorize them, such as semantic methods vs syntactic methods, prior/posterior methods vs differential methods, etc. See surveys [1][2][3] for details. Another way to categorize privacy definitions is by whether *non-independent reasoning (NIR)* is considered as a privacy violation. In NIR, the information about one record in the data can be learnt from the information of other records in the data, under the assumption that these records follow the same underlying distribution. A classifier is a master example of NIR where the class information of a new instance is learnt from the distribution in a related training set.

Most posterior/prior based privacy definitions consider NIR as a privacy violation, such as l-diversity [4], t-closeness [5], ρ_1-ρ_2 privacy [6], β-likeness [7], *small sum privacy* [8] and Δ-growth [9]. These criteria quantify the risk to an individual by the information learnt from the subpopulation containing that individual. To avoid privacy violation, the information learnt is required to have a small change compared to a prior of an adversary, and this often requires to "smooth" the distribution in the published data. One drawback of this approach is that it is hard to model the prior of the attacker [10][11]. Another drawback is that it limits the desired utility of learning statistical relationships. For example, Δ-growth postulates that the distribution in each subpopulation should be close to the global distribution in the whole data set. This requirement makes it difficult to learn novel statistical relationships such as "smokers tend to have lung cancer" in the subpopulation of smokers.

At the other side of the aisle, differential privacy [10] considers NIR as a non-privacy violation, as stated in [11] (page 4): "We explicitly consider non-independent reasoning as a non-violation of privacy; information that can be learned about a row from sources other than the row itself is not information that the row could hope to keep private". Instead of avoiding the occurrence of disclosures, the differential privacy criterion seeks to mask the impact of a single individual on such occurrences. A popularized claim is that, even if an attacker knows all but one records, the attacker will not learn much about the remaining tuple. As indicated above, this comes with the price of permitting disclosures through NIR. Indeed, the recent study in [12] suggests that disclosures could occur under differential privacy if records are correlated, and the study in [13] demonstrates that a Bayes classifier could be built using only differentially private answers to predict the sensitive attribute of an individual. In this paper we propose that a sensitive disclosure of NIR could occur in more general cases: no correlation among

10.5441/002/edbt.2015.41

Table 1: {Prof-school, Prof-specialty, White, Male} → >50K (Conf=83.83%)

	$\epsilon = 0.01$ ($b = 200$)		$\epsilon = 0.1$ ($b = 20$)		$\epsilon = 0.5$ ($b = 4$)	
	Mean	SE	Mean	SE	Mean	SE
$Conf'$	1.34392	1.36299	0.860966	0.0985138	0.832659	0.0645165
$\lvert ans_1 - ans'_1\rvert/ans_1$	0.614742	0.533185	0.0693353	0.0272098	0.0262412	0.0144438
$\lvert ans_2 - ans'_2\rvert/ans_2$	0.570118	0.983959	0.102247	0.0820627	0.069974	0.0636316

records is required and only two differentially private query answers are needed to infer the sensitive attribute value. The example below demonstrates such a disclosure.

EXAMPLE 1. *Consider the* ADULT *data set [14] that contains 45,222 records (without missing values) from the 1994 Census database. We did not observe any record correlation in this data set. Consider the five attributes Education, Occupation, Race, Gender, and Income. The Income attribute has two values, "≤50K", for 75.22% of records, and ">50K", for 24.78% of records. We assume that learning the Income value for a record is sensitive. On the raw data, the following two count queries Q_1 and Q_2 return the answers $ans_1 = 501$ and $ans_2 = 420$, respectively:*

Q_1: "Prof-school ∧ Prof-specialty ∧ White ∧ Male",
Q_2: "Prof-school ∧ Prof-specialty ∧ White ∧ Male ∧ >50K".

These answers imply the following rule with the confidence $Conf = \frac{ans_2}{ans_1} = 0.8383$.

{Prof-school, Prof-specialty, White, Male} → >50K.

Since this confidence is significantly higher than the overall frequency 24.78% of the value ">50K", this rule may violate the privacy of the individuals matching the condition of Q_1. While this rule seems expected, it does demonstrate the potential risk of NIR on a real life data distribution. After all, truly sensitive data and findings are difficult to obtain and publish.

The differential privacy mechanism will return the noisy answers $ans'_i = ans_i + \xi_i$, $i = 1, 2$, where the noises ξ_i's follow some distribution, and an adversary has to gauge $Conf$ by $Conf' = \frac{ans'_2}{ans'_1}$. Consider the widely used Laplace noise distribution $Lap(b) = \frac{1}{2b} exp(-\frac{|\xi|}{b})$, where b is the scale factor. The setting of $b = \Delta/\epsilon$ would ensure ϵ-differential privacy for the sensitivity Δ of the query function. Let us set $\Delta = 2$ to account for the two count queries. Note that the effect of a larger Δ can be simulated by the effect of a smaller ϵ because $b = \Delta/\epsilon$.

Table 1 shows the mean of $Conf'$ and the relative error $\frac{|ans_i - ans'_i|}{ans_i}$ of query answers over 10 trials of random noises, and the standard error (SE) of the mean. $Conf'$ measures the disclosure (in red) and $\frac{|ans_i - ans'_i|}{ans_i}$ measures the utility of query answers (in blue). At the higher privacy level $\epsilon = 0.01$, $Conf'$ deviates substantially from $Conf = 0.8383$, but the utility of the noisy answers is also poor because of the large relative errors and SE. At the lower privacy level $\epsilon = 0.5$, the utility of noisy answers improves significantly, but $Conf' = 0.8327$ is within 1% difference from $Conf$ with a small SE (i.e., 0.0645); in this case any instances of ans'_1 and ans'_2 are sufficient to gauge the income level of an individual. □

To ensure a good utility, a fixed (and small) scale b of noises is essential. Indeed, improving utility through reducing noises is a major focus of the work on differential privacy (see [15] for a list). As the query answer becomes larger, such noises become less significant, which improves the utility of noisy answers ans'_i, therefore, the accuracy of $\frac{ans'_2}{ans'_1}$. Thus, the good utility of ans'_i comes together with the risk of disclosures. A general and quantitative analysis on this type of attack will be presented in Section 2. Choosing a large noise scale (i.e., a smaller ϵ) helps thwart such attacks, but it also hurts the utility for data analysis. In fact, as long as the noise scale stays fixed, the noises eventually become insignificant for large query answers.

1.2 Our Approach

The question we study in this paper is how to (A) allow learning statistical relationships (such as "smokers tend to have lung cancer"), and at the same time, (B) prevent learning sensitive information about an individual (such as "Bob likely has HIV"). As discussed above, posterior/prior based privacy criteria provide (B) but not (A), whereas the differential privacy criterion provides (A) but not (B). The difficulty of providing (A) and (B) is that they both make use of NIR, one for utility and one for privacy violation. The key lies at distinguishing these two types of learning. The next example illustrates the ideas of our approach.

EXAMPLE 2. *Consider a table $D(Gender, Job, Disease)$, where Gender and Job are public and Disease is sensitive. Assume that Disease has 10 possible values. To hide the Disease value, for each record in D, uniform perturbation [16] for a given retention probability, say 20%, will retain the Disease value in the record with 20% probability and replace it with a value chosen uniformly from the 10 possible values of Disease at random with the remaining 80% probability. This can be implemented by tossing a biased coin with head probability 20%. Let D^* denote the perturbed data.*

D^* can be utilized to reconstruct the distribution of Disease in a given subset of records. Consider any subset S of D, the counterpart S^* for D^*, and any Disease value d. Let f_d denote the (actual) frequency of d in S, f^*_d denote the (observed) frequency of d in S^*, and $E[F^*_d]$ denote the expectation of f^*_d (over all coin tosses). All frequencies are in fraction. The following equation follows from the perturbation operation applied to the data:

$$E[F^*_d] = (0.2 + 0.8/10) f_d + (0.8/10)(1 - f_d) \quad (1)$$

Approximating the unknown $E[F^*_d]$ by the observed f^*_d, we get an estimate of f_d as $\frac{f^*_d - 0.08}{0.2}$. This estimate is the maximum likelihood estimator (MLE) *[16] computed using the perturbed S^*.*

Given the published D^*, suppose that an adversary tries to learn the likelihood that Bob, a male engineer with a record in D, has breast cancer or BC for short. One way is considering the subset S_{me} for all male engineers in D, and another is considering the subset S_e for all engineers in D. Let M^{me}_d and M^e_d be the MLE for a disease d in S_{me} and S_e, respectively. Two questions can be asked.

Question 1: Which of M^{me}_{BC} and M^e_{BC} should be used to quantify the risk to Bob? *S_{me} contains exactly the records that match all Bob's public information, whereas S_e contains additional records that do not belong to Bob. Without further information, S_{me} is more relevant to Bob than S_e, so M^{me}_{BC} should be used as the risk to Bob. If the additional records for female engineers follow a different distribution on BC from those for male engineers, M^e_{BC} most likely is not useful for inferring whether Bob has breast* *cancer. We will discuss the case where the additional records have the same distribution as Bob in Section 3.4. On the other hand, the frequency M_d^e for some disease d (e.g., cervical spondylosis) may be useful for data analysis, such as learning the statistical relationship that career engineers tend to have d. This leads to the next question.*

Question 2: How to limit the accuracy of M_{BC}^{me} while preserving the accuracy of M_d^e for data analysis? *The errors of M_d^{me} and M_d^e were caused by approximating the unknown $E[F_d^*]$ with the observed f_d^* in Equation (1). From the law of large numbers, f_d^* is closer to $E[F_d^*]$ when more records are randomized (i.e., more coin toss). Since S_e^* contains more records than S_{me}^*, M_d^e is more accurate for estimating the frequency of d in S_e than M_d^{me} for estimating the frequency of d in S_e. We can leverage this gap to limit the accuracy of M_{BC}^{me} while preserving the accuracy of M_d^e.* □

This example illustrates two types of reconstruction for MLEs. The reconstruction of M_{BC}^{me} based on S_{me} is called *personal reconstruction* because it aims at a particular individual by matching all public attributes of Bob; the reconstruction of M_d^e based on S_e is called *aggregate reconstruction* because it aims at a large population without specifically targeting any individual. We argue (in Section 3.2) that personal reconstruction is the source of privacy concerns whereas aggregation reconstruction is the source of utility. The law of large numbers suggests that these two types of reconstruction respond differently to the reduction of record perturbation. We leverage this gap to limit the accuracy of personal reconstruction while preserving the accuracy of aggregate reconstruction.

The small count privacy and large count utility in [8] use the number of records involved to distinguish the reconstruction for privacy concern and the reconstruction for utility. It is not clear how to set appropriate thresholds for such sizes. Indeed, it could be the case that two reconstructions are performed on two subsets of data with the same size but one aims at finding an individual's sensitive information while the other aims at finding general patterns.

1.3 Contributions

Here are the main contributions in this work:

Contribution 1 (Section 2): We present a condition to characterize the occurrence of disclosures of differentially private answers through NIR. For the Laplace noise distribution, this condition is simple and neat as it is expressed in terms of the ratio of the scale factor to the query answer.

Contribution 2 (Section 3): We propose an *inaccuracy requirement* on personal reconstruction as a new privacy criterion called *reconstruction privacy*. This criterion imposes a minimum value δ for the *best* upper bound on $\Pr\left[\frac{F'-f}{f} > \lambda\right]$ for the actual and estimated frequency f and F' of a sensitive value in a personal reconstruction, where δ and λ are privacy parameters. Note that $\frac{F'-f}{f}$ is the error of the reconstruction for f, which should *not* be confused with the relative increase of the attacker's belief such as the β-likeness [7], (n,t)-closeness [17] and $(c,2)$-diversity [4]. This criterion does not bound the maximum value of F' or f or require them to be close to the global distribution, making it suitable for learning statistical relationships through aggregate reconstruction. Also, this criterion avoids modeling the prior of an adversary, which can be tricky as shown in [10][11].

Contribution 3 (Sections 4): We present an efficient test of reconstruction privacy. First, we show a conversion between an upper bound for the tail probability of Poisson trials into an upper bound on $\Pr\left[\frac{F'-f}{f} > \lambda\right]$. Then, we obtain an efficient test of reconstruction privacy by adapting the notion of reconstruction privacy to an existing upper bound for Poisson trials, i.e., the Chernoff bound.

Contribution 4 (Section 5): We present an efficient algorithm for producing a perturbed version D^* that satisfies a given specification of reconstruction privacy. The algorithm is highly efficient because it only needs to sort the records once and make another scan on the sorted data.

Contribution 5 (Section 6): We evaluate two claims. The first claim is that reconstruction privacy could be violated by real life data sets even after data perturbation. The second claim is that the proposed method can preserve utility for statistical learning while providing reconstruction privacy.

2. OBSERVATIONS ON DIFFERENTIAL PRIVACY

In this section, we answer the question under what conditions would differentially private answers disclose sensitive information through NIR? The standard ϵ-differential privacy mechanism [10] ensures that, for any two data sets D_1 and D_2 differing on at most one record, for all queries Q of interest, and for any value α in the range for noisy answers, $\Pr[K(D_1,Q) = \alpha] \leq exp(\epsilon)\Pr[K(D_2,Q) = \alpha]$, where $K(D_i,Q)$ is a noisy answer $a+\xi$ for the actual answer a and a random noise ξ following some distribution. The *scale* $E[|\xi|]$ of noises depends on the query class and the privacy parameter ϵ. The purpose of the noise is to mask the impact of a single record on query answers.

Let us construct a disclosure by differentially private answers. Let SA denote the sensitive attribute (e.g., diseases) and NA denote the public attributes. Suppose that an adversary tries to determine whether a participating individual t has a particular value sa on SA. Let $t.NA$ denote the values for t on NA, which is known to the adversary. The adversary issues two count queries:

$$\begin{aligned} Q_1 :& NA = t.NA \\ Q_2 :& NA = t.NA \wedge SA = sa \end{aligned} \tag{2}$$

Let $X = x + \xi_1$ and $Y = y + \xi_2$ be the noisy answers for Q_1 and Q_2 returned by the ϵ-differential privacy mechanism, where x and y are actual answers and ξ_i's are the noises. Note $\frac{y}{x} \leq 1$ and $\frac{y}{x}$ represents the chance that t has the sa value on SA. Note $\frac{Y}{X} = \frac{y+\xi_2}{x+\xi_1} = \frac{y/x+\xi_2/x}{1+\xi_1/x}$.

The intuition that $\frac{Y}{X}$ may lead to a disclosure is as follows. For any ξ_i of a fixed scale, as the answer x increases, ξ_2/x and ξ_1/x decrease and $\frac{Y}{X}$ approaches $\frac{y}{x}$. If $\frac{y}{x}$ is large enough (which is application specific), the adversary learns that t has the sensitive value sa with a high probability. This construction is general because it does not assume record correlation and does not depend on the noise distribution except that the noises have a fixed scale. Below, we formalize this intuition. First, we show a lemma.

LEMMA 1. *Let x and y be the true answers to Q_1 and Q_2, $x \neq 0$. Let $X = x + \xi_1$ and $Y = y + \xi_2$ be the noisy answers for Q_1 and Q_2 with the noises ξ_i having the zero mean and the variance V. Then*

$$E[\tfrac{Y}{X}] \simeq \tfrac{y}{x}(1+\tfrac{V}{x^2}) \textit{ and } Var[\tfrac{Y}{X}] \simeq \tfrac{V}{x^2}(1+\tfrac{y^2}{x^2})$$

PROOF. Note that $E[\frac{Y}{X}]$ is *not* equal to $\frac{E[Y]}{E[X]}$. Using the Taylor expansion technique [18, 19], $E[\frac{Y}{X}]$ and $Var[\frac{Y}{X}]$ can be approximated as follows:

$$E[\frac{Y}{X}] \simeq \frac{E[Y]}{E[X]} + \frac{cov[X,Y]}{E[X]^2} + \frac{Var[X]E[Y]}{E[X]^3}$$

$$Var[\frac{Y}{X}] \simeq \frac{Var[Y]}{E[X]^2} - \frac{2E[Y]}{E[X]^3}cov[X,Y] + \frac{E[Y]^2}{E[X]^4}Var[X]$$

The error of the approximation is the remaining terms of the Taylor expansion that are dropped. $E[X] = x$ and $E[Y] = y$ (because noises have the zero mean), $Var[X] = Var[Y] = V$, and the covariance $cov[X,Y] = cov[x+\xi_1, y+\xi_2] = cov[\xi_1, \xi_2]$. Since ξ_1 and ξ_2 are unrelated, $cov[\xi_1, \xi_2] = 0$. Substantiating these into the above equations and simplifying, we get $E[\frac{Y}{X}]$ and $Var[\frac{Y}{X}]$ as required. $\square$

For any noise distribution with the zero mean and a fixed variance V, as the query answer x increases, $\frac{V}{x^2}$ decreases, $E[\frac{Y}{X}]$ approaches $\frac{y}{x}$ and $Var[\frac{Y}{X}]$ approaches zero. In general, $E[\frac{Y}{X}]$ approaching $\frac{y}{x}$ does not entail $\frac{Y}{X}$ approaching $\frac{y}{x}$, for particular instances X and Y. However, if $Var[\frac{Y}{X}]$ approaches zero, the deviation of $\frac{Y}{X}$ from $E[\frac{Y}{X}]$ approaches zero, $\frac{Y}{X}$ approaches $\frac{y}{x}$. This is summarized in the next corollary.

COROLLARY 1. *For any noise distribution with the zero mean and a fixed variance V, as the query answer x increases, $\frac{Y}{X}$ approaches $\frac{y}{x}$.*

To our knowledge, Corollary 1 covers all noise distributions employed by the differential privacy mechanism, including Laplace mechanism [10], Gaussian mechanism [20], and Matrix mechanism [21], because these distributions have a zero mean and a fixed variance. To see how large x is needed for $\frac{Y}{X}$ to be accurate enough for $\frac{y}{x}$, let us consider the Laplace mechanism $Lap(b) = \frac{1}{2b}exp(-|\xi|/b)$, but a similar analysis can be performed for other mechanisms. b is the *scale factor*. $Lap(b)$ has the zero mean and the variance $V = 2b^2$. The setting $b = \Delta/\epsilon$ ensures ϵ-differential privacy, where Δ is the *sensitivity* of the queries of interest, which roughly denotes the worst-case change in the query answer on changing one record in any possible database. Δ is a property of the queries, not a property of the database. Hence, V is fixed for a given query class and Corollary 1 applies to $Lap(b)$. Substituting $\frac{y}{x} \leq 1$ and $\frac{V}{x^2} = \frac{2b^2}{x^2} = 2\left(\frac{b}{x}\right)^2$ into Lemma 1 and simplifying, we get a simple bound on $|E[\frac{Y}{X}] - \frac{y}{x}|$ and $Var[\frac{Y}{X}]$ in terms of the scale factor b and the query answer x (but not the privacy parameter ϵ or the sensitivity Δ of queries).

COROLLARY 2. *Let X and Y be the noisy answers for actual answers x and y, where the noises follow the Laplace distribution $Lap(b)$. (i) $|E[\frac{Y}{X}] - \frac{y}{x}| \leq 2\left(\frac{b}{x}\right)^2$. (ii) $Var[\frac{Y}{X}] \leq 4\left(\frac{b}{x}\right)^2$.*

Table 2: $2\left(\frac{b}{x}\right)^2$

x / b	5000	1000	500	200	100
$b = 10$ ($\epsilon = 0.2$)	**0.000008**	**0.0002**	**0.0008**	**0.005**	0.02
$b = 20$ ($\epsilon = 0.1$)	**0.000032**	**0.0008**	**0.0032**	0.02	0.08
$b = 40$ ($\epsilon = 0.05$)	**0.000128**	**0.0032**	0.0128	0.08	0.32
$b = 200$ ($\epsilon = 0.01$)	**0.0032**	0.08	0.32	2	8

Thus, the value of $2\left(\frac{b}{x}\right)^2$ is an indicator of how close $\frac{Y}{X}$ is to $\frac{y}{x}$. Table 2 shows the values of $2\left(\frac{b}{x}\right)^2$ for various query answers x and settings of b (within the brackets is the corresponding privacy parameter ϵ for the setting of $\Delta = 2$, which accounts for answering the two queries Q_1 and Q_2 in a row). The boldface highlights where $2\left(\frac{b}{x}\right)^2$ is small enough so that $\frac{Y}{X}$ is a good indicator of $\frac{y}{x}$. Take ($b = 20, x = 500$) as an example where $2\left(\frac{b}{x}\right)^2 = 0.0032$. $|E[\frac{Y}{X}] - \frac{y}{x}| \leq 0.0032$ and $Var[\frac{Y}{X}] \leq 0.0032 \times 2 = 0.0064$. Indeed, Corollary 2 quantifies a condition of the occurrence of disclosures in terms of $\frac{b}{x}$: as a rule of thumb, a ratio $\frac{b}{x} \leq \frac{1}{20}$ would ensure that $\frac{Y}{X}$ is a good indicator of $\frac{y}{x}$ because $2\left(\frac{b}{x}\right)^2 \leq \frac{2}{400}$. In this case, if $\frac{y}{x}$ is high enough to be considered as sensitive, a sensitive disclosure would occur through accessing noisy answers X and Y. This condition also suggests that such disclosures cannot be avoided by choosing a large scale factor b if the actual answer x can be arbitrarily large.

We end this section with an explicit acknowledgement of disclosures by differential privacy from [10]: "Note that a bad disclosure can still occur, but our guarantee assures the individual that it will not be the presence of her data that causes it, nor could the disclosure be avoided through any action or inaction on the part of the user". In the rest of the paper, we present an approach to avoid the disclosures of NIR in a data perturbation approach. This effort can be considered as an action on the part of the data publisher.

3. PROBLEM STATEMENT

We define our model of data perturbation, privacy criterion, and the problems we will study.

3.1 Data Perturbation

As in [7, 9, 22], we consider a table D that has one sensitive (private) attribute denoted by SA and several pubic attributes denoted by $NA = \{A_1, \cdots, A_n\}$. We assume that the domain of SA has $m > 2$ sensitive values, $sa_1, \cdots, sa_m$.

Assumptions. To hide the SA information of a record, we perturb the SA value but keep the attributes in NA unchanged in a record. We assume that an adversary has no prior knowledge on positive correlation between NA and SA; otherwise, the public information on NA already discloses the information about SA. The adversary can have prior knowledge on correlation among the attributes in NA, which presents no problem because we never modify the attributes in NA. We also assume that an adversary has no prior knowledge about correlation among SA of *different* records. This assumption can be satisfied by including exactly one record from a set of correlated records, as suggested in [23].

Prior knowledge on negative correlation [24] deserves some more explanations. Consider the negative correlation "females do not have prostate cancer". This correlation tells that the observed prostate cancer is not the original SA value for a female, but does not tell what is the original value because each of the remaining $m-1$ values has an equal probability. For this reason, we assume that m is larger than 2 (or even larger) so that guessing a remaining value has enough uncertainty. We should emphasize that this situation is not unique for data perturbation, and differentially private answers have similar issues: if the noisy answer for the query on "Female and Prostate Cancer" is -5 (or more generally, too small according to prior knowledge), the above negative correlation would disclose a small range of the noise added, i.e., -5 or less, after observing the noisy answer, which invalids the Laplace distribution assumption. In general, if too much information is leaked through prior knowledge, no mechanism will work.

One criticism on distinguishing SA and NA is that such distinction can be tricky sometimes. This deserves some clarification as well. One approach that does not make such distinction is treating all attributes as sensitive attributes and randomizing a record over the Cartesian product of the domains of all attributes [25][23]. Unfortunately, this approach is vulnerable to undoing the randomization by removing "infeasible" records added during randomization. An example of infeasible records is (Age=1, Job=prof, Disease=HIV) since a 1-year child can not possibly be a professor, so the adversary can easily tell that this record was added by randomization. Treating Age and Job as public attributes and randomizing only Disease can avoid this problem. In general, treating and randomizing more attributes like sensitive ones when they are actually public attributes would introduce more vulnerabilities to the removal of "infeasible" records. In this sense, randomizing only the truly sensitive attribute actually provides more protection.

We produce the perturbed version D^* of D by applying *uniform perturbation* [25][16][6] on SA as follows. For a given retention probability p, where $0 < p < 1$, for each record in D, we toss a coin with head probability p. If the coin lands on head, retain the SA value in the record; if the coin lands on tail, replace the SA value in the record with a value picked from the domain of SA with equal probability (i.e., $\frac{1-p}{m}$) at random. This perturbation operator is characterized by the following matrix $\mathbb{P}_{m\times m}$:

$$\mathbb{P}_{ji} = \begin{cases} p + \frac{1-p}{m} & \text{if } j{=}i \text{ (retain } sa_i\text{)} \\ \frac{1-p}{m} & \text{if } j{\neq}i \text{ (perturb } sa_i \text{ to } sa_j\text{)} \end{cases} \quad (3)$$

A proper choice of the retention probability p can ensure some privacy requirements, such as ρ_1-ρ_2 privacy [6][25]. We end this section with a comparison between *output perturbation* and *data perturbation* in the current work. In output perturbation, such as the differential privacy approach, a noise is added to the query answer and the noisy answer is used *as is*. For this reason, a *small* and *fixed* noise scale is essential for good utility. As discussed in Sections 1.1 and 2, as the data size increases, such noises are vulnerable to NIR. In data perturbation, the SA value in each record is perturbed independently and the original distribution of SA must be *reconstructed* from the perturbed records by taking into account the perturbation operation performed. As the data size increases, the number of record perturbation increases *proportionally*, which is less vulnerable to NIR. In addition, data perturbation is more amendable to record insertion because each record is perturbed independently and the reconstruction is performed by the user himself. In contrast, updating (published) noisy query answers can be tricky because a new record could affect multiple queries and a correlated change of query answers can be exploited by the adversary to learn the information about the new record.

3.2 Types of Reconstruction

We adopt the following notation. Let $NA = \{A_1, \cdots, A_n\}$. For $1 \leq i \leq n$, let x_i be either a domain value of A_i or a wildcard, denoted by $-$, that matches every domain value of A_i. $D(x_1, \cdots, x_n)$ denotes the subset of records in D that match x_i on every A_i, and $D^*(x_1, \cdots, x_n)$ denotes the corresponding subset for D^*. If, for $1 \leq i \leq n$, x_i is a non-wildcard, $D(x_1, \cdots, x_n)$ is a *personal group*. If at least one x_i is a wildcard, $D(x_1, \cdots, x_n)$ is an *aggregate group*. For example, for $NA = \{Gender, Job\}$, $D(male, eng)$ is a personal group and $D(-, eng)$ is an aggregate group. Intuitively, a personal group contains all records that can not be distinguished by any information other than SA. For example, even if an adversary may know the age of Bob, this information is not helpful to distinguish any record in the personal group $D(male, eng)$ because all records in the personal group are exactly identical on NA. Without confusion, we call both $D(x_1, \cdots, x_n)$ and $D(x_1, \cdots, x_n)^*$ a personal or aggregate group as there is an one-to-one correspondence between the two.

In Example 2, we argued that the personal group $D^*(male, eng)$ should be used to quantify the risk of inferring the disease breast cancer for the male engineer Bob, instead of the aggregate groups $D^*(-, eng)$, $D^*(male, -)$, or $D^*(-, -)$. The rationale is that unless further information is available, it is to the adversary's advantage not to use a record that is *known* not belonging to Bob. In Section 3.4 we will consider the case where further information is available to the adversary and using additional records not belonging to Bob may help the adversary. An analogy is short-listing the suspect of a robbery: if the eyewitness has reported that the suspect was a male blonde caucasians (i.e., the public attributes), it makes sense to focus on the subset of male blonde caucasians in the police database, instead of examining all male caucasians records. The above observation motivates the following two types of reconstruction.

DEFINITION 1. *A* personal reconstruction *refers to estimating the frequencies of the SA values in a personal group g based on the perturbed g^*. An* aggregate reconstruction *refers to estimating the frequencies of the SA values in an aggregate group g based on the perturbed g^*.* □

We consider a personal reconstruction as the source of privacy concern because it aims specifically at an individual by matching all the individual's public information. In contrast, we consider an aggregate reconstruction as the source of utility because it aims at a larger population without specifically targeting a particular individual. These different roles of reconstruction are stated in the next principle.

DEFINITION 2 (SPLIT ROLE PRINCIPLE). *A personal reconstruction aims specifically at a particular individual and is responsible for privacy violation. An aggregate reconstruction aims at a larger population and is responsible for providing utility. As far as privacy protection is concerned, it suffices to ensure that personal reconstruction is not accurate.* □

Remarks. The Split Role Principle provides only a relative privacy guarantee because some disclosure can still occur to an individual through aggregate reconstruction in the name of utility, such as "females tend to have breast cancer (compared to males)". But our principle assures the individual that such disclosures are not specifically targeting him or her, and those that do (i.e., personal reconstruction) have been made unreliable. In fact, any statistical database with any non-trivial utility incurs some amount of disclosure [10]. Our principle assures that only a limited amount of disclosure is incurred by enabling non-trivial utility.

3.3 Reconstruction Privacy

Under the Split Role Principle, our privacy guarantee is that all personal reconstructions are not effective for learning the information about SA. To formalize this guarantee, consider a personal group g^* and g, and a particular SA value sa. Let f denote the frequency of sa in g and let F' denote the estimate of f obtained from the personal reconstruction based on g^*. Note that F' is a random variable because D^* is a result of coin tosses. $\frac{F'-f}{f}$ is the relative error of F'. A larger $\frac{F'-f}{f}$ means that an adversary faces more uncertainty in using F' to gauge of the likelihood of sa for an individual. The next definition formalizes an "inaccuracy requirement" on $\frac{F'-f}{f}$.

DEFINITION 3 (RECONSTRUCTION PRIVACY). *Let $\lambda > 0$ and $\delta \in [0, 1]$. sa is (λ, δ)*-reconstruction-private *in a personal group g^* if $\Pr\left[\frac{F'-f}{f} > \lambda\right] < U$ or $\Pr\left[\frac{F'-f}{f} < -\lambda\right] < L$, for some U and L, implies $\delta \leq min\{U, L\}$. A personal group g^* is (λ, δ)*-reconstruction-private *if every sa is (λ, δ)*-reconstruction-private *in g^*. D^* is (λ, δ)*-reconstruction-private *if every personal group g^* is (λ, δ)-reconstruction-private. (All probabilities are taken over the space of coin tosses during the perturbation of SA values.)* □

Note that reconstruction privacy is a property of the perturbation matrix $\mathbb{P}$, not a property of a particular instance of D^*. In plain words, (λ, δ)-reconstruction-privacy ensures that the *smallest upper bound* is not less than δ; in this sense, the adversary has difficulty to get an accurate estimate of f, and the larger λ or δ is, the greater this difficulty is. As an example, violating $(0.3, 0.3)$-reconstruction-privacy by g^* means that the adversary can get a smaller-than-0.3 upper bound on $\Pr\left[\frac{F'-f}{f} > 0.3\right]$ or $\Pr\left[\frac{F'-f}{f} < -0.3\right]$. This implies at least one of the following:

$$\Pr\left[\frac{F'-f}{f} \leq 0.3\right] \geq 70\%, \text{ where } F' > f$$
$$\Pr\left[\frac{F'-f}{f} \geq -0.3\right] \geq 70\%, \text{ where } F' < f$$

Our definition considers such a high probability of a small error as a potential risk.

Remarks. $F' - f$ should not be confused with the change in the posterior belief of an adversary. In fact, f is the probability of sa in the personal group g and F' is the estimate of f based on the personal reconstruction for g^*, and $\frac{F'-f}{f}$ is the relative error of the estimate. Our definition considers a small estimation error as a privacy risk, regardless of the absolute value of f, on the basis that any accurate person reconstruction is potentially a risk because it discloses the actual distribution of SA that aims at a target individual. The choice of the relative error, instead of the absolute error, is necessary because a larger actual frequency f requires a larger absolute error for protection. Bounding the accuracy of estimating f, instead of bounding the posterior belief of an adversary, has two important benefits: it allows the room for learning statistical relationships (through aggregate reconstruction), and it frees the publisher of measuring the adversary's prior belief and specifying a threshold for posterior beliefs, which can be tricky [10][11]. Finally, the choice of smallest upper bounds, rather than lower bounds, on $\Pr\left[\frac{F'-f}{f} > \lambda\right]$ and $\Pr\left[\frac{F'-f}{f} < -\lambda\right]$, allows us to leverage the literature on upper bounds for random variables to estimate $\Pr\left[\frac{F'-f}{f} > \lambda\right]$.

DEFINITION 4 (ENFORCING PRIVACY). *Given a database D, a retention probability p $(1 > p > 0)$ for perturbing SA, and privacy parameters λ and δ, devise an algorithm that enforces (λ, δ)-reconstruction-privacy on D^* while preserving aggregate reconstruction as much as possible.* □

By leaving the retention probability p as an input parameter to our problem, other privacy criteria, such as ρ_1-ρ_2 privacy, can be enforced through a proper choice of p. In this sense, reconstruction privacy can be considered as an *additional* protection on top of other privacy criteria.

3.4 Generalized Personal Groups

Consider two personal groups $g^* = D(male, eng)$ and $g'^* = D(female, eng)$. Our reconstruction privacy limits the reconstruction for each personal group, but does not limit the reconstruction for the combined $g^* \cup g'^*$, i.e., the aggregate group $D^*(-, eng)$, because the reconstruction for $g^* \cup g'^*$ is not relevant to an individual, assuming that males and females have a different distribution on SA, such as on breast cancer. However, this argument may be invalid if the adversary has further knowledge about the distribution of SA values. For example, suppose that $FavoriteColor$ is another public attribute and that the favorite color of an individual has nothing to do with the diseases, the adversary may do reconstruction after aggregating all personal groups that differ only in the values on $FavoriteColor$, and such reconstruction is more accurate than the reconstruction based on a single personal group because it uses more randomized records. In this case, aggregate groups disclose sensitive information.

To address this issue, for each public attribute A_i, if two domain values x_i and x'_i (e.g., $male$ and $female$) of A_i have the same impact on SA, we will merge x_i and x'_i into a single generalized value, and we define personal groups based on such generalized values. With this preprocessing, every generalized value of A_i now has a different impact on SA, thus, has a different distribution on SA. Then our previous argument that an aggregate group does not provide a representative statistics for a target individual remains valid because such groups combine several sub-populations that follow a different distribution on SA.

So the question is how to identify the values of A_i that have the same impact on SA. To this end, the well studied χ^2-squared test that tells if two data sets are from different distributions can be used. For two domain values x_i and x'_i of A_i, let o_{ij} (resp. o'_{ij}) be the number of records in D satisfying $A_i = x_i$ (resp. $A_i = x'_i$) and $SA = sa_j$, $1 \leq j \leq m$. Let $O_i = [o_{i1}, \cdots, o_{im}]$ and $O'_i = [o'_{i1}, \cdots, o'_{im}]$, which represents the distributions of SA conditioned on x_i and x'_i. In proper statistical language, can we disprove, to a certain required level of significance, the *null hypothesis* that the two data sets O_i and O'_i are drawn from the same population distribution function? Disproving the null hypothesis in effect proves that the data sets are from different distributions.

Since $|O_i| = \sum_{j=1}^{m} o_{ij}$ and $|O'_i| = \sum_{j=1}^{m} o'_{ij}$ are not necessarily equal, our case is that of two binned distributions with unequal number of data points. In this case, the degree of freedom is equal to m and the χ^2 value is computed as [26]:

$$\chi^2 = \sum_{j=1}^{m} \frac{\left(\sqrt{|O'_i|/|O_i|}o_{ij} - \sqrt{|O_i|/|O'_i|}o'_{ij}\right)^2}{o_{ij} + o'_{ij}} \qquad (4)$$

Then we obtain the expected value of χ^2 by checking the chi-square distribution with two parameters, the degree of freedom (e.g., m) and the value of *significance*, the maximum probability that the computed χ^2 from Equation (4) could be greater than the expected χ^2. We set the conventional setting of 0.05 for significance. If the value computed by Equation (4) is greater than this expected value of χ^2, we can disprove the null hypothesis that the two data sets O_i and O'_i are drawn from the same population distribution function because the probability for this is less than 5% (i.e., the significance). Otherwise, we consider that the two data sets are consistent with a single distribution function.

We represent the χ^2 test results for all pairs (x_i, x'_i) of values of A_i using a graph. Each value x_i of A_i is a vertex in the graph and we connect two vertices x_i and x'_i if the χ^2 test on (x_i, x'_i) fails to disprove the above null hypothesis. Finally, for each connected component of the graph, we merge all the values in the component into a single generalized value. This method ensures that any two values x_i and x'_i from different components have a different impact on SA.

In the rest of the paper, we assume that the domain values of each public attribute A_i are generalized values produced by the above merging procedure and that the personal and aggregate groups defined in Section 3.2 are based on such generalized domain values.

4. TESTING PRIVACY

An immediate question is how to test (λ, δ)-reconstruction-privacy. From Definition 3, this requires to obtain the smallest upper bounds U and L on $\Pr\left[\frac{F'-f}{f} > \lambda\right]$ and $\Pr\left[\frac{F'-f}{f} < -\lambda\right]$. The follow-

Table 3: Notations

Symbols	Meaning
D, D^*	the raw data and perturbed version
S, S^*	a subset of records and perturbed version
g, g^*	a personal group and perturbed version
m	the domain size $\|SA\|$
t	a target individual
sa_i	a domain value of SA
f_i	the frequency of sa_i in S
o_i^*	the count of sa_i in S^*
O_i^*	the variable for o_i^*
F_i'	the variable for the estimate of f_i
$\overleftarrow{f}, \overleftarrow{F'}, \overleftarrow{O^*}$	the column-vectors of f_i, F_i', O_i^*
$\mathbb{P}$	the perturbation matrix in Equation (3)
p	the retention probability
(λ, δ)	privacy parameters

ing discussion refers to a subset S of D and the corresponding subset S^* of D^*. $|S|$ denotes the number of records in S. Let $(f_1, \cdots, f_m)$ be the frequencies of SA values $(sa_1, \cdots, sa_m)$ in S, $(O_1^*, \cdots, O_m^*)$ be the variables for the observed counts of $(sa_1, \cdots, sa_m)$ in S^*, and $(F_1', \cdots, F_m')$ be the variables for an estimate of $(f_1, \cdots, f_m)$ reconstructed using S^*. These vectors are also written as column-vectors $\overleftarrow{f}$, $\overleftarrow{O^*}$, and $\overleftarrow{F'}$. When no confusion arises, we drop the subscripts i from f_i, O_i^*, F_i'. Table 3 summarizes the notations used in this paper.

4.1 Computing F'

First of all, let us examine the computation of F'. Example 2 illustrates the basic idea of computing the estimate F' of f for a particular SA value sa based on the perturbed data. Generalizing that idea to the vectors $\overleftarrow{F'}$ and $\overleftarrow{f}$, our perturbation operation implies the equation $\mathbb{P} \cdot \overleftarrow{f} = \frac{E[\overleftarrow{O^*}]}{|S|}$, where $\mathbb{P}$ is the perturbation matrix in Equation (3). Approximating $E[\overleftarrow{O^*}]$ by the observed counts $\overleftarrow{O^*}$, we get the estimate of $\overleftarrow{f}$ given by $\mathbb{P}^{-1} \cdot \frac{\overleftarrow{O^*}}{|S|}$, where $\mathbb{P}^{-1}$ is the inverse of $\mathbb{P}$. This estimate is called the *maximum likelihood estimator* (MLE).

THEOREM 1 (THEOREM 2, [16]). *$\mathbb{P}^{-1} \cdot \frac{\overleftarrow{O^*}}{|S|}$ is the MLE of $\overleftarrow{f}$ under the constraint that its elements sum to 1. Let $\overleftarrow{F'}$ denote this MLE.* □

The next lemma gives an equivalent computation of $\overleftarrow{F'}$ without referring to $\mathbb{P}^{-1}$.

LEMMA 2. *For any subset S of D and any SA value sa, (i) $E[O^*] = |S|(fp + (1-p)/m)$, (ii) $F' = \frac{O^*/|S| - (1-p)/m}{p}$, and (iii) $E[F'] = f$.*

PROOF. (i) O^* comes from two sources of records in S: those that have the SA value sa and are retained, and those that have a SA value other than sa and are perturbed to sa. The expected number of the records in the first source is $|S|f(p+(1-p)/m)$, and the expected number of the records in the second source is $|S|(1-f)(1-p)/m)$. Summing up the two gives $E[O^*] = |S|(fp+(1-p)/m)$. This shows (i).

(ii) From Theorem 1, $\overleftarrow{F'} = \mathbb{P}^{-1} \cdot \frac{\overleftarrow{O^*}}{|S|}$. Let $\overleftarrow{\frac{1-p}{m}}$ denote the column-vector of the constant $\frac{1-p}{m}$ of length m. We have

$$\frac{\overleftarrow{O^*}}{|S|} = \mathbb{P} \cdot \overleftarrow{F'} = p\overleftarrow{F'} + \overleftarrow{\frac{1-p}{m}}$$

Thus, $F' = \frac{O^*/|S| - (1-p)/m}{p}$, as required for (ii).

(iii) Taking the mean on both sides of the last equation, $E[F'] = \frac{E[O^*]/|S| - (1-p)/m}{p}$. Substituting $E[O^*]$ in (i) and simplifying, we get $E[F'] = f$. This shows (iii). □

Lemma 2(iii) implies that F' is an unbiased estimator of f. Lemma 2(ii) gives a computation of F' in terms of the known values O^*, $|S|$, p, m without referring to $\mathbb{P}^{-1}$. In the rest of the paper, we adopt this computation of F' in the definition of reconstruction privacy (Definition 3).

4.2 Bounding $\Pr\left[\frac{F'-f}{f} > \lambda\right]$ and $\Pr\left[\frac{F'-f}{f} < -\lambda\right]$

Recall that $F' = \frac{O^*/|S| - (1-p)/m}{p}$ from Lemma 2(ii). To bound $\Pr\left[\frac{F'-f}{f} > \lambda\right]$ and $\Pr\left[\frac{F'-f}{f} < -\lambda\right]$, we first obtain the upper bounds for the error of *observed* O^* and then convert them into the upper bounds for the error of *reconstructed* F'. The next theorem gives the conversion between these bounds.

THEOREM 2 (BOUND CONVERSION). *Consider any subset S of D and any SA value sa with the frequency f in S. Let O^* be the observed count of sa in S^* and let F' be the MLE of f. Let $\mu = E[O^*]$. For any functions $U(\omega, \mu)$ and $L(\omega, \mu)$ of ω and μ, and for a comparison operator $\bigoplus$ that is either $<$ or $>$,*

1. *$\Pr\left[\frac{O^*-\mu}{\mu} > \omega\right] \bigoplus U(\omega, \mu)$ if and only if $\Pr\left[\frac{F'-f}{f} > \lambda\right] \bigoplus U(\omega, \mu)$;*
2. *$\Pr\left[\frac{O^*-\mu}{\mu} < -\omega\right] \bigoplus L(\omega, \mu)$ if and only if $\Pr\left[\frac{F'-f}{f} < -\lambda\right] \bigoplus L(\omega, \mu)$.*

where $\lambda = \frac{\omega\mu}{|S|pf}$.

PROOF. We show 1) only because the proof for 2) is similar. From $F' = \frac{O^*/|S| - (1-p)/m}{p}$ (Lemma 2(ii)), $O^* = |S|(F'p + (1-p)/m)$, and from Lemma 2(i), $\mu = |S|(fp + (1-p)/m)$. So $\frac{O^*-\mu}{\mu} > \omega \Leftrightarrow O^* - \mu > \omega\mu \Leftrightarrow |S|p(F'-f) > \omega\mu \Leftrightarrow \frac{F'-f}{f} > \frac{\omega\mu}{|S|pf}$. 1) follows by letting $\lambda = \frac{\omega\mu}{|S|pf}$. □

According to Theorem 2, if we have the smallest upper bounds on $\Pr\left[\frac{O^*-\mu}{\mu} > \omega\right]$ or $\Pr\left[\frac{O^*-\mu}{\mu} < -\omega\right]$, we immediately have the smallest upper bounds on $\Pr\left[\frac{F'-f}{f} > \lambda\right]$ or $\Pr\left[\frac{F'-f}{f} < -\lambda\right]$. This conversion does not hinge on the particular form of the bound functions U and L, and applies to both upper bounds (when $\bigoplus$ is $<$) and lower bounds (when $\bigoplus$ is $>$). Therefore, finding the smallest upper bounds for F' is reduced to that for O^*. The latter can benefit from the literature on upper bounds for tail probabilities of Poisson trials. Markov's inequality and Chebyshev's inequality are some early upper bounds, for example. The Chernoff bound, due to [27], is a much tighter bound as it gives exponential fall-off of probability with distance from the error. The following is a simplified yet tight form of the Chernoff bound.

THEOREM 3 (CHERNOFF BOUNDS, [27]). *Let $X_1, \cdots, X_n$ be independent Poisson trials such that for $1 \le i \le n$, $X_i \in \{0, 1\}$, $\Pr[X_i = 1] = p_i$, where $0 < p_i < 1$. Let $X = X_1 + \cdots + X_n$ and $\mu = E[X] = E[X_1] + \cdots + E[X_n]$. For $\omega \in (0, \infty)$,*

$$\Pr\left[\frac{X-\mu}{\mu} > \omega\right] < U(\omega, \mu) = exp(-\frac{\omega^2\mu}{2+\omega}) \quad (5)$$

and for $\omega \in (0, 1]$,

$$\Pr\left[\frac{X-\mu}{\mu} < -\omega\right] < L(\omega, \mu) = exp(-\frac{\omega^2\mu}{2}).\square \quad (6)$$

The observed count O^* of sa in S^* is equal to $X = X_1 + \cdots + X_n$, where X_i is the indicator variable whether the i-th row in S^* has the value sa. If the i-th row has sa prior to perturbation, $p_i = p + (1-p)/m$, otherwise, $p_i = (1-p)/m$. $E[O^*] = |S|(fp + (1-p)/m)$ (Lemma 2). To obtain the upper bounds for F', we instantiate the upper bounds U and L for O^* in Equations (5) and (6) into Theorem 2. This gives the next corollary.

COROLLARY 3 (UPPER BOUNDS FOR F'). *Let* $\omega = \frac{\lambda|S|pf}{\mu}$ *and* $\mu = |S|(fp + (1-p)/m)$. *For* $\omega \in (0, \infty)$,

$$\Pr\left[\frac{F'-f}{f} > \lambda\right] < U(\omega,\mu) = exp(-\frac{\omega^2\mu}{2+\omega}) \tag{7}$$

and for $\omega \in (0, 1]$,

$$\Pr\left[\frac{F'-f}{f} < -\lambda\right] < L(\omega,\mu) = exp(-\frac{\omega^2\mu}{2}).\square \tag{8}$$

Note that $\omega = \frac{\lambda pf}{pf+(1-p)/m}$ and $\mu = |S|(fp + (1-p)/m)$. λ, p, f, m are constants. Reducing $|S|$ decreases μ, which increases the upper bounds U and L *exponentially*. Thus, reducing $|S|$ effectively thwarts the attacker from bounding $\Pr\left[\frac{F'-f}{f} > \lambda\right]$ and $\Pr\left[\frac{F'-f}{f} < -\lambda\right]$ by a small upper bound. Our enforcement algorithm presented in the next section is based on this observation.

A remaining question is whether $U = exp(-\frac{\omega^2\mu}{2+\omega})$ and $L = exp(-\frac{\omega^2\mu}{2})$ in Corollary 3 derived from the Chernoff bound for O^* are the smallest upper bounds for F', as required by the definition of (λ,δ)-reconstruction-privacy. Suppose not. There would exist a smaller upper bound U_2 on $\Pr\left[\frac{F'-f}{f} > \lambda\right]$ or a smaller upper bound L_2 on $\Pr\left[\frac{F'-f}{f} < -\lambda\right]$. Then Theorem 2 implies that U_2 and L_2 are better bounds than the Chernoff bounds U and L for O^*. However, the fact that the Chernoff bound remained in use in the past 60 years suggests that finding smaller upper bounds is difficult. Until the Chernoff bound is improved, we assume that the upper bounds U and L in Corollary 3 are the best upper bounds for F'. This assumption is not a real restriction because Theorem 2 allows us to "plug in" any better bound for O^* for a better bound for F'. If the adversary finds a better bound than the Chernoff bound and the data publisher still uses the Chernoff bound. If the better bound is a general result and the publisher refuses to "plug in" it, the responsibility is with the publisher. Otherwise, under our assumptions about prior knowledge in Section 3.1, getting a better bound requires knowledge about the random coin tosses in the perturbation process. Like all randomized mechanisms, we assume that actual results of random trails are not available to the adversary.

4.3 Testing

With the upper bounds L and U in Corollary 3, it is straightforward to test whether (λ,δ)-reconstruction-privacy holds by testing $\delta \le min\{L, U\}$. We can further simplify this test. For ω in the range $(0, 1]$, it is easy to see $L < U$, therefore, $\delta \le min\{L, U\}$ degenerates into $\delta \le L$. Substituting the expressions for ω and μ in Corollary 3 into $L(\omega,\mu)$, we get $L = exp(-\frac{(\lambda pf)^2|S|}{2(fp+(1-p)/m)})$, where λ is in the range $(0, 1 + \frac{(1-p)/m}{pf}]$, which corresponds to the range $(0, 1]$ for ω. Substituting the expression for L into $\delta \le L$ gives rise to the following test of (λ,δ)-reconstruction-privacy.

COROLLARY 4. *Let sa be a SA value, g be a personal group, and f be the frequency of sa in g. For* $\lambda \in (0, 1 + \frac{(1-p)/m}{pf}]$ *and* $\delta \in [0, 1]$, *sa is* (λ,δ)*-reconstruction-private in* g^* *if and only if*

$$|g| \le \frac{-2(fp + (1-p)/m)\ln\delta}{(\lambda pf)^2}\square \tag{9}$$

Given D, the personal groups g and the frequencies f for all SA values in g can be found by sorting the records in D in the order of all attributes in NA followed by SA. Therefore, all the quantities in Equation (9) are either given (i.e., λ, δ, p, m) or can be computed efficiently (i.e., f and $|g|$). A larger $|g|, f, p$ makes this inequality less likely hold, thus, makes (λ,δ)-reconstruction-privacy more likely violated. In fact, under these conditions there are either more random trials or more retention of the SA value, which leads to a more accurate reconstruction.

5. ENFORCING PRIVACY

If reconstruction privacy is not satisfied, we can restore reconstruction privacy by satisfying the condition in Equation (9) for every SA value and every personal group. Observe that the right-hand side of Equation (9) decreases as f increases. Therefore, a personal group g^* satisfies reconstruction privacy if and only if $|g| \le s_g$, where

$$s_g = \frac{-2(fp + (1-p)/m)\ln\delta}{(\lambda pf)^2} \tag{10}$$

and f is the maximum frequency for any SA value in g. Another interpretation is that s_g is the maximum number of independent trials if g^* is to satisfy reconstruction privacy. If $|g| > s_g$, reconstruction privacy is violated (because of too many independent trails). To fix this, one approach is increasing s_g to the current group size $|g|$ by reducing f or p (note that m, λ, δ are fixed). This approach is not preferred because reducing f will distort the data distribution and reducing p has a global effect of making the perturbed data too noisy. Our approach is reducing $|g|$ to the size s_g by *sampling* a subset g_1 of the size s_g and *perturbing* g_1 instead of g. This sampling essentially reduces the excessive number of independent random trials. To ensure $s_{g_1} = s_g$, g_1 must preserve the (relative) frequency of every SA value in g (to the right-hand side of Equation (10) unchanged after sampling). Preserving frequencies also helps minimize the distortion to data distribution. After perturbing the sample g_1, a *scaling* step is needed to scale the perturbed g_1^* back to the original size $|g|$ to minimize the impact on the global distribution. Below, we present an algorithm named *Sampling-Perturbing-Scaling (SPS)* to meet both the group size requirement and the frequency preservation requirement.

Sampling-Perturbing-Scaling (SPS) algorithm. The input is a database D, the retention probability p $(0 < p < 1)$, the domain size m of SA, and the privacy parameters λ and δ. The output is a modified version of D^* that satisfies (λ,δ)-reconstruction-privacy. For each personal group g in D, this algorithm computes a modified version g_2^* of g^*, then outputs $D_2^* = \bigcup g_2^*$. In a preprocessing step, we sort the records in D by the attributes in NA and followed by SA. The result is a collection of personal groups g together with the frequencies f of every SA value in g.

For each personal group g in D, compute s_g as in Equation (10), if $|g| \le s_g$, g already satisfies the maximum group size constraint, let $g_2^* = g^*$. We assume $|g| > s_g$. In the following, g_2^* is produced in three steps: *Sampling*, *Perturbing*, and *Scaling*, described below. Let $\tau = s_g/|g|$, called the *sampling rate*.

1. $Sampling(g, s_g)$ takes a sample of the records in g while preserving the frequency of each SA value. For each SA value sa occurring in g, let g_{sa} denote the subset of the records

in g that have sa. Note that all records in g_{sa} are identical. We pick any $\lfloor |g_{sa}|\tau \rfloor$ records from g_{sa} and pick one additional record from g_{sa} with the probability $|g_{sa}|\tau - \lfloor |g_{sa}|\tau \rfloor$. Let g_1 be the set of the picked records. Return g_1.

2. $Perturbing(g_1, p, m)$ perturbs the SA values of the records in g_1 with the retention probability p, as in the Uniform Perturbation described in Section 3.1. Return g_1^*.

3. $Scaling(g_1^*, |g|)$ scales up g_1^* to the original size $|g|$ while preserving the frequency of each SA value. Let $\tau' = |g|/|g_1^*|$. For each record r^* in g_1^*, let g_2^* contain $\lfloor \tau' \rfloor$ duplicates of r^* and one additional duplicate of r^* with the probability $\tau' - \lfloor \tau' \rfloor$. Return g_2^*.

Remarks. Several points are worth noting. First, *Sampling* kicks in only if $|g|$ exceeds the maximum size s_g; otherwise, all records in g will be used for perturbation. Therefore, if the data set is small enough to have such a poor accuracy that already satisfies reconstruction privacy, our algorithm will behave like the standard uniform perturbation without performing sampling. In this case, the poor accuracy is not caused by our sampling, but by the inadequate amount of data. Second, the duplication in *Scaling* does not introduce new random trials because it is performed *after* the perturbation in g_1^*. The adversary may notice some duplicate records in g_2^*, but this is not a problem because privacy is actually achieved on g_1^* before the scaling step.

Complexity analysis. Let $|D|$ denote the number of records in D. The sorting step takes $|D|log|D|$ time to generate all personal groups. Subsequently, each of the steps $Sampling$, $Perturbing$, and $Scaling$ takes one data scan. A more efficient implementation, however, is to perform these three steps in a single data scan: as a record r is sampled, immediately we perturb the SA value of r and then duplicate the perturbed record a certain number of times as described, and add the duplicates to g_2^*. In total, the algorithm takes $(|D|log|D| + |D|)$ time.

5.1 Analysis

We prove two claims about the output $D_2^* = \cup g_2^*$. The first claim is on privacy guarantee: each g_2^* in D_2^* is (λ, δ)-reconstruction-private. The second claim is on utility: for any subset S consisting of one or more personal groups and the corresponding subset S_2^* in D_2^*, F'_{g_2} is an unbiased estimator of f, where f is the frequency of a particular SA value in S and F'_{g_2} is the estimate of f reconstructed from S_2^*, respectively. We first present some facts.

Let g be a personal group. Assume $|g| > s_g$. Let g_1, g_1^*, g_2^* be computed for g and let $O_g^*, O_{g_1}^*, O_{g_2}^*$ be the observed count for a particular SA value sa in g^*, g_1^*, g_2^*, respectively. Let f_g and f_{g_1} be the frequency of sa in g and g_1. Let F'_g, F'_{g_1}, F'_{g_2} be the MLE reconstructed from g^*, g_1^*, g_2^*. We avoid to use f_1, F'_1, F'_2 as these symbols have been used as the frequencies for SA values sa_1 and sa_2. Let $u \simeq v$ denote that u and v are equal modulo the random trial for the additional record in *Scaling* and *Sampling*.

- Fact 1: $f_{g_1} \simeq f_g$ and $|g_1| \simeq s_g$. This is because *Sampling* preserves the frequency of sa in g and the sample g_1 has the size s_g.

- Fact 2: $O_{g_2}^*/|g_2^*| \simeq O_{g_1}^*/|g_1^*|$. This is because *Scaling* from g_1^* to g_2^* preserves the frequency of sa.

- Fact 3: $F'_{g_1} \simeq F'_{g_2}$. This follows from $F'_{g_i} = \frac{O_{g_i}^*/|g_i^*| - (1-p)/m}{p}$, $i = 1, 2$ (Lemma 2(ii)) and Fact 2.

- Fact 4: $E[O_{g_2}^*] \simeq E[O_g^*]$. From Lemma 2(i), $E[O_{g_1}^*] = |g_1|(f_{g_1}p + (1-p)/m) \simeq s_g(f_{g_1}p + (1-p)/m)$ (Fact 1). Since *Scaling* duplicates each record in g_1^* by $\frac{|g|}{s_g}$ times, $E[O_{g_2}^*] \simeq \frac{|g|}{s_g} \times E[O_{g_1}^*] = |g|(f_{g_1}p + (1-p)/m)$. From Lemma 2(i), $E[O_g^*] = |g|(f_g p + (1-p)/m)$. Then $f_{g_1} \simeq f_g$ (Fact 1) implies $E[O_{g_2}^*] \simeq E[O_g^*]$.

THEOREM 4 (PRIVACY). *For each personal group g, g_2^* returned by the SPS algorithm is (λ, δ)-reconstruction-private.*

PROOF. If $|g| \leq s_g$, $g_2^* = g^*$, by Corollary 4, g_2^* is (λ, δ)-reconstruction-private. We assume $|g| > s_g$. In this case, g_1^* is (λ, δ)-reconstruction-private because $|g_1| \simeq s_g$ (Fact 1). We claim $\frac{F'_{g_2} - f_g}{f_g} \simeq \frac{F'_{g_1} - f_{g_1}}{f_{g_1}}$, which implies that F'_{g_2} has the same tail probability for error as F'_{g_1}; therefore, g_2^* is (λ, δ)-reconstruction-private because g_1^* is. This claim follows from $f_{g_1} \simeq f_g$ (Fact 1) and $F'_{g_1} \simeq F'_{g_2}$ (Fact 3). $\square$

THEOREM 5 (UTILITY). *Let S be a set of records for one or more personal groups in D, S^* be the corresponding set for D^*, and S_2^* be the corresponding set for D_2^*. Let f be the frequency of a SA value sa in S, and let F' and F'_{S_2} be the estimates of f reconstructed from S^* and S_2^*. Then $E[F'_{S_2}] \simeq f$.*

PROOF. Let $O_2^* = \sum O_{g_2}^*$, $O^* = \sum O_g^*$, $|S^*| = \sum |g^*|$, and $|S_2^*| = \sum |g_2^*|$, where $\sum$ is over the personal groups g for S. $|S^*| \simeq |S_2^*|$. From Lemma 2(ii), $E[F'] = \frac{E[O^*]/|S^*| - (1-p)/m}{p}$ and $E[F'_{S_2}] = \frac{E[O_2^*]/|S_2^*| - (1-p)/m}{p}$. From Fact 4, $E[O^*] \simeq E[O_2^*]$. Thus, $E[F'] \simeq E[F'_{S_2}]$. From Lemma 2(iii), $E[F'] \simeq f$, thus, $E[F'_{S_2}] \simeq f$. $\square$

Intuitively, Theorem 5 says that the estimate reconstructed using the corresponding records in D_2^* is an unbiased estimator of the actual frequency.

6. EXPERIMENTAL STUDIES

We evaluate two claims. The first claim is that reconstruction privacy could be violated on real life data sets. The second claim is that the proposed SPS algorithm eliminates personal reconstruction with minor sacrifice on the utility of aggregate reconstruction.

6.1 Experimental Setup

We implemented the proposed SPS algorithm as described in Section 5 in C++ and ran all experiments on an Intel Xeon(R) E5630 CPU 2.53GHZ PC with 12GB of RAM. We utilized two publicly available data sets. The first one is the *ADULT* data set [14]. This data set has 45,222 records (without missing values) extracted from the 1994 Census database with the attributes Education, Occupation, Race, Gender, and Income. We chose Income as SA and the remaining attributes as the public attributes NA. The second data set is the *CENSUS* data previously used in [28][22]. This data set contains personal information about 500K American adults with 6 discrete attributes Age, Gender, Education, Marital, Race, and Occupation. We chose Occupation as SA and the remaining attributes as NA. We considered five samples of CENSUS of sizes $100K$, $200K$, $300K$, $400K$, $500K$. These data sets have different characteristics: ADULT represents a small data set with very few SA values (with Income having only two values), whereas CENSUS represents a large data set with a large number of balanced distributed SA values (with Occupation having 50 values). We want to see how these characteristics would affect the evaluation of our claims.

As discussed in Section 3.4, the values for public attributes with the same impact on SA have to be aggregated before generating personal groups. The aggregation affects data sets to some extent. Tables 4 and 5 show the impacts on the domain size of each public attribute, the total number of personal groups (e.g., $|G|$), and the averaged personal groups size (e.g., $|D|/|G|$ with $|D|$ as the total number of records) of ADULT and CENSUS $300K$. In the rest of this section, we use the generalized values of public attributes.

Table 4: NA Aggregation Impact on ADULT

	Domain Size of NA				$\vert G\vert$	$\vert D\vert/\vert G\vert$
	Education	Occupation	Race	Gender		
Before Aggregation	16	14	5	2	2240	20
After Aggregation	7	4	2	2	112	404

Table 5: NA Aggregation Impact on CENSUS $300K$

	Domain Size of NA					$\vert G\vert$	$\vert D\vert/\vert G\vert$
	Age	Gender	Education	Marital	Race		
Before Aggregation	77	2	14	6	9	116424	3
After Aggregation	1	2	14	6	9	1512	331

The utility of the published data is evaluated by the accuracy of answering count queries of the form:

$$\begin{aligned}&SELECT\ COUNT\ (*)\ FROM\ D\\&WHERE\ A_1 = a_1 \wedge \cdots \wedge A_d = a_d \wedge SA = sa_i\end{aligned} \tag{11}$$

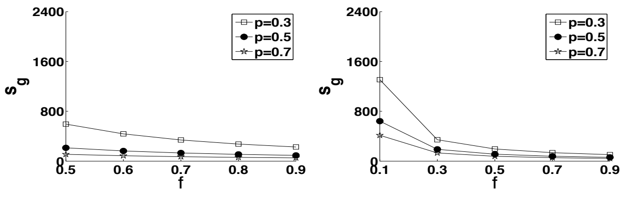
where $A_j \in NA$, $a_j \in dom(A_j)$, and $sa_i \in dom(SA)$. The answer to the query, ans, is the number of records in D satisfying the condition in the WHERE clause. Such answers can be used to learn statistical relationships between the attributes in NA and SA. Given the perturbed data D^*, ans is approximated by $est = |S^*| * F'$, where S^* is the set of records in D^* satisfying $A_1 = a_1 \wedge \cdots \wedge A_d = a_d$, $|S^*|$ is the size of S^*, and F' is the MLE given by Lemma 2(ii) based on S^*. The *relative error* of est is defined as $\frac{|est-ans|}{ans}$. A smaller relative error means a larger accuracy and better utility. Queries on only NA are not considered because such queries have zero relative error.

Data mining and analysis typically focuses on low dimensional statistics, such as 1D or 2D marginals with a size above a sanity bound [29]. We generated a pool of 5,000 count queries with the query dimensionality d in $\{1, 2, 3\}$ and with the selectivity $ans/|D| \geq 0.1\%$. For each query, we selected d from $\{1,\ 2,\ 3\}$, selected d attributes from NA without replacement, selected a value $a_i \in dom(A_i)$ for each selected attribute A_i, and finally selected a value $sa_i \in dom(SA)$. All selections are random with equal probability. If the query's selectivity is 0.1% or more, we replaced the NA value with aggregated values and then added it to the pool. Recall that we aggregated NA values based on their impact on SA as in Section 3.3. The query pool simulates the set of possible queries generated from real life, therefore, the original NA value (before aggregation) is used to generate the query pool. Since we protect reconstruction privacy on aggregated personal groups we evaluate relative error on these aggregated personal groups as well. We report the average of relative error over all queries in this pool. In addition, since D^* is randomly generated in each run, we reported the average of 10 runs to avoid the bias of a particular run.

Table 6: Parameter Table

Parameters	**Settings**
p	0.1, 0.3, **0.5**, 0.7, 0.9
λ	0.1, 0.2, **0.3**, 0.4, 0.5
δ	0.1, 0.2, **0.3**, 0.4, 0.5

The uniform perturbation, denoted by UP, as described in Section 3.1 has been used as a privacy mechanism in [25][16][6]. But these privacy mechanisms do not address the disclosure of personal reconstruction. Our method addresses this disclosure by applying UP to sampled data. So our evaluation has two parts. First, we evaluate how often reconstruction privacy is violated by the perturbed data D^* produced by UP. Then, we evaluate the cost of achieving reconstruction reconstruction by our SPS algorithm. This cost is measured by the increase in the relative error for queries answered using D_2^* produced by SPS, compared to the relative error of queries answered using D^* produced by UP. The same retention probability p is used for both UP and SPS. Table 6 shows the settings of p, λ, and δ with the default settings in boldface.



(a) ADULT (b) CENSUS

Figure 1: Maximum group size s_g vs. maximum frequency f

Below, a group means a personal group. First, we study the condition $|g| \leq s_g$ for testing whether a group g^* satisfies reconstruction-privacy as described in Section 5, where s_g is the maximum threshold on the group size defined as

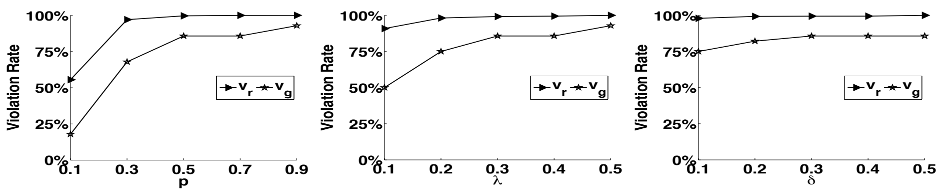
$$s_g = \frac{-2(fp + (1-p)/m)\ln\delta}{(\lambda pf)^2} \tag{12}$$

f is the maximum frequency of any SA value occurring in g. Figure 1 plots the relationship between s_g and f (for the default settings of λ and δ). Note that the range of f is $[0.5, 0.9]$ for ADULT, but is $[0.1, 0.9]$ for CENSUS. This is because ADULT contains only 2 distinct SA values, as a result, f is at least 50% in all personal groups. Each curve corresponds to a setting of p. For each curve in Figure 1, the region above the curve represents the area where this condition fails, that is, $|g| > s_g$ for a given f. The large area above these curves suggests that the maximum group size s_g can be easily exceeded, and thus, there is a good chance of violating reconstruction privacy. Observing both Figure 1 and Equation (12) we get that, when parameters: λ, δ and p are given, the value of m and f have opposite effects on the value of s_g, particularly, f becomes the dominant factor when f is small (e.g., when $f \leq 0.3$ in Figure 1). The value of s_g boosts when f is smaller, implying that personal groups with smaller f tend to be reconstruction private because it is easier for them to satisfy the condition of $|g| \leq s_g$. We will confirm this observation on the two real life data sets shortly.

6.2 ADULT Data Set

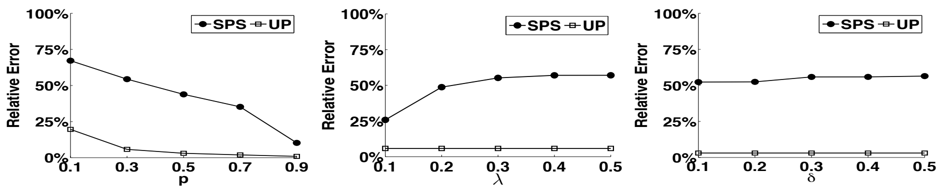
Violation. Figure 2 shows the extent to which reconstruction privacy is violated on the perturbed ADULT data set D^* produced by UP. This *extent* is measured at two levels. v_g represents the percentage of groups that violate reconstruction privacy. v_r represents the percentage of records contained in a violating personal group, i.e., the coverage of the violating groups in terms of the number of individuals affected. We consider this coverage because all the records in a violating group are under the same risk of accurate personal reconstruction.

Both violations in terms of v_r and v_g are obvious. Take the default setting of $p = 0.5$, $\lambda = 0.3$ and $\delta = 0.3$ as an example. The



(a) vs. p (b) vs. λ (c) vs. δ

Figure 2: ADULT: Privacy Violation



(a) vs. p (b) vs. λ (c) vs. δ

Figure 3: ADULT: Relative Error

85% of all groups are violating and covering more than 99% of the records. This privacy risk is interpreted as follows: with probability of $1 - \delta = 70\%$, the estimate F' of some SA value is within a relative error of $\lambda = 30\%$, and this case covers more than $v_r = 99\%$ of all individuals. The large coverage is expected because a larger group more likely violates reconstruction privacy (Figure 1).

Cost. Figure 3 shows the increase of relative error due to the sampling of SPS. Compared to UP, the relative error for SPS increases about 50% in the *worst case*. This increase is due to the sampling required to eliminate the violation of reconstruction privacy. Considering the large coverage of the violation (i.e., v_r in Figure 2), having such increase of error is reasonable. We emphasize that this increase is due to the large f in personal groups in ADULT. Recall that f is no less than 50% and when f is larger personal groups tend to violate reconstruction privacy (Figure 1). Note that ADULT is not general in real life in terms of very few number of SA values, for other data sets with more SA values, the increased error would be reduced, which will be confirmed soon on the CENSUS data set. Choosing a small p helps eliminate violation, but also quickly increases the relative error for both UP and SPS (Figures 2a and 3a). Indeed, a too small p makes the perturbed data become nearly pure noises. This study confirms our discussion at the beginning of Section 5 that the approach of reducing p does not preserve utility.

6.3 CENSUS Data Set

Violation. CENSUS is a larger data set with a much larger number of balanced distributed SA values. We are curious how this characteristic change would affect our claims. Figure 4 shows the extent to which reconstruction privacy is violated. The default data size is 300K when $|D|$ is not specified. Compared to the ADULT data set, the frequency f of a SA value is much smaller; consequently, the value of s_g is much larger (Figure 1). The larger s_g makes it easy to satisfy the condition of $|g| \leq s_g$, therefore, it is less likely that groups in CENSUS would violate reconstruction privacy, which explains the much smaller v_g and also confirms our claim on Figure 1 that smaller f may lead to less reconstruction violations. Besides, the larger s_g implies that violation groups must have larger g because $|g| > s_g$, which explains the small number of violation groups covering the most records in the data set.

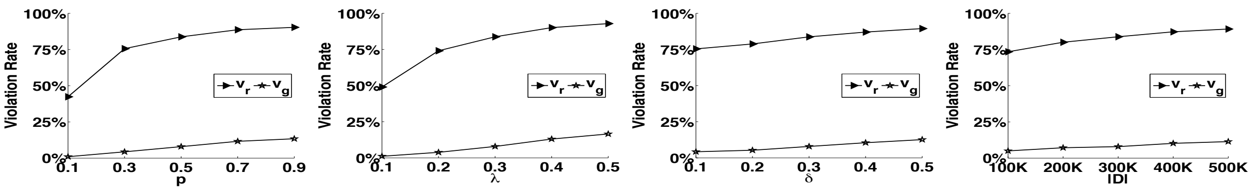
Cost. Figure 5 compares the relative error of UP and SPS. A big difference from the ADULT data set is that there is less increase in the relative error (e.g., less than 10% for most of settings) for SPS compared to the relative error for UP across all settings of parameters. This is a consequence of the smaller percentage r_g of the violating groups discussed above. In this case, most of the groups do not need sampling because they satisfy reconstruction privacy and only the small number of violating groups will be sampled. Even for such groups, a small reduction in the number of record perturbation is sufficient to increase the error of personal reconstruction to the level required by our privacy criterion.

Another interesting point is that even though a larger data size $|D|$ causes more violations of reconstruction privacy (Figure 4d), it actually decreases the relative error for SPS (Figure 5d). As explained above, for this data set, eliminating violation incurs little additional error beyond that of UP. Therefore, as the data size increases, the relative error of UP gets smaller, so does the relative error of SPS. This finding suggests that the proposed SPS algorithm could be more effective on a larger data set.

In summary, our empirical studies supported the claim that reconstruction attack could occur on real life data sets, whether they are small or large and whether the number of sensitive attribute is small or large. The studies also supported the claim that the proposed privacy criterion and the sampling method are effective to preserve the utility for data analysis while eliminating such attacks. This effectiveness is more observed on larger data sets with a large number of balanced distributed sensitive attributes.

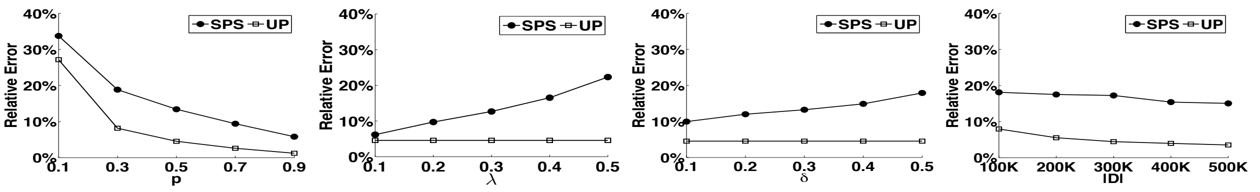
7. CONCLUSION

Differential privacy has become a popular privacy definition for sharing statistical information thanks to good utility. However, this good utility comes with the cost of disclosures through non-independent reasoning. In this work, we presented a data perturbation approach to prevent sensitive non-independent reasoning while enabling statistical learning. We achieved these goals through a property implied by the law of large numbers, which allows us to separate these two types of learning by their different responses to reduction in random trials. Based on this idea, we use record sampling to reduce the random trials in data perturbation, which mostly affects non-independent reasoning specific to an individual while having only a limited effect on statistical learning.



(a) vs. p (b) vs. λ (c) vs. δ (d) vs. $|D|$

Figure 4: CENSUS: Privacy Violation



(a) vs. p (b) vs. λ (c) vs. δ (d) vs. $|D|$

Figure 5: CENSUS: Relative Error

8. ACKNOWLEDGEMENTS

Ke Wang's work is partially supported by a Discovery Grant of the Natural Sciences and Engineering Research Council, Canada. Raymond Chi-Wing Wong's is partially supported by the grant F-SGRF14EG34. Philip S. Yu's is partially supported by US NSF through grants CNS-1115234 and OISE-1129076.

9. REFERENCES

[1] R. Adam and J. Worthmann. Security-control methods for statistical databases: A comparative study. *ACM Comput. Surv.*, 21(4):515–556, December 1989.

[2] B. Fung, K. Wang, R. Chen, and P. Yu. Privacy-preserving data publishing: a survey of recent developments. *ACM Comput. Surv.*, 42(4):14:1–14:53, June 2010.

[3] B. Chen, D. Kifer, K. LeFevre, and A. Machanavajjhala. Privacy-preserving data publishing. *Found. Trends databases*, 2(1-2):1–167, January 2009.

[4] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam. L-diversity: privacy beyound k-anonymity. In *ICDE*, 2006.

[5] N. Li, T. Li, and S. Venkatasubramanian. t-closeness: privacy beyond k-anonymity and l-diversity. In *ICDE*, 2007.

[6] A. Evfimievski, J. Gehrke, and R. Srikant. Limiting privacy breaches in privacy preserving data mining. In *PODS*, 2003.

[7] J. Cao and P. Karras. Publishing microdata with a robust privacy guarantee. In *VLDB*, 2012.

[8] A. Fu, K. Wang, R. Wong, J. Wang, and M. Jiang. Small sum privacy and large sum utility in data publishing. *Journal of Biomedical Informatics*, 50:20–31, 2014.

[9] Y. Tao, X. Xiao, J. Li, and D. Zhang. On anti-corruption privacy preserving publication. In *ICDE*, 2008.

[10] C. Dwork. Differential privacy. In *ICALP*, 2006.

[11] A. Blum, C. Dwork, F. McSherry, and K. Nissim. Practical privacy: the sulq framework. In *PODS*, 2005.

[12] D. Kifer and A. Machanavajjhala. No free lunch in data privacy. In *SIGMOD*, 2011.

[13] G. Cormode. Personal privacy vs population privacy: learning to attack anonymization. In *SIGKDD*, 2011.

[14] Adult data set. http://archive.ics.uci.edu/ml/datasets/Adult.

[15] C. Li. *Optimizing liner queries under differential privacy*. PhD thesis, Computer Science, University of Massachusetts Amherst, 2013.

[16] R. Agrawal, R. Srikant, and D. Thomas. Privacy preserving olap. In *SIGMOD*, 2005.

[17] M. NarasimhaRao, J. VenuGopalkrisna, R. Murthy, and C. Ramesh. Closeness: Privacy measure for data publishing using multiple sensitive attributes. 2(2):278–284, 2012.

[18] R. C. Elandt-Johnson and N.L. Johnson. *Survival models and data analysis*. John Wiley & Sons NY, 1980.

[19] A. Stuart and K. Ord. *Kendall's advanced theory of statistics*, volume 1. Arnold, London, 6 edition, 1998.

[20] C. Dwork, K. Kenthapadi, F. McSherry, I. Mironov, and M. Naor. Our data, ourselves: privacy via distributed noise generation. In *EUROCRYPT*, volume 4004, pages 486–503, 2006.

[21] C. Li, M. Hay, V. Rastogi, G. Miklau, and A. McGregor. Optimizing linear counting queries under differential privacy. In *PODS*, 2010.

[22] R. Chaytor and K. Wang. Small domain randomization: same privacy, more utility. In *VLDB*, 2010.

[23] V. Rastogi, S. Hong, and D. Suciu. The boundary between privacy and utility in data publishing. In *VLDB*, 2007.

[24] T. Li and N. Li. Injector: mining background knowledge for data anonymization. In *ICDE*, 2008.

[25] S. Agrawal and J. Haritsa. A framework for high-accuracy privacy preserving mining. In *ICDE*, 2005.

[26] W. Press, B. Flannery, S. Teukolsky, and W. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 1988.

[27] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Mathematical Statistics*, 23(4):493–507, 1952.

[28] X. Xiao and Y. Tao. Anatomy: simple and effective privacy preservation. In *VLDB*, 2006.

[29] X. Xiao, G. Bender, M. Hay, and J. Gehrke. ireduct: Differential privacy with reduced relative errors. In *SIGMOD*, 2011.

Time series anomaly discovery with grammar-based compression

Pavel Senin
University of Hawaiʻi at Mānoa
Collaborative Software Development Laboratory
senin@hawaii.edu

Jessica Lin, Xing Wang
George Mason University
Dept. of Computer Science
jessica@gmu.edu, xwang24@gmu.edu

Tim Oates, Sunil Gandhi
University of Maryland, Baltimore County
Dept. of Computer Science
oates@cs.umbc.edu, sunilga1@umbc.edu

Arnold P. Boedihardjo   Crystal Chen   Susan Frankenstein
U.S. Army Corps of Engineers, Engineer Research and Development Center
{arnold.p.boedihardjo, crystal.chen, susan.frankenstein}@usace.army.mil

ABSTRACT

The problem of anomaly detection in time series has recently received much attention. However, many existing techniques require the user to provide the length of a potential anomaly, which is often unreasonable for real-world problems. In addition, they are also often built upon computing costly distance functions – a procedure that may account for up to 99% of an algorithm's computation time.

Addressing these limitations, we propose two algorithms that use grammar induction to aid anomaly detection without any prior knowledge. Our algorithm discretizes continuous time series values into symbolic form, infers a context-free grammar, and exploits its hierarchical structure to effectively and efficiently discover algorithmic irregularities that we relate to anomalies. The approach taken is based on the general principle of Kolmogorov complexity where the randomness in a sequence is a function of its algorithmic incompressibility. Since a grammar induction process naturally compresses the input sequence by learning regularities and encoding them compactly with grammar rules, the algorithm's inability to compress a subsequence indicates its Kolmogorov (algorithmic) randomness and correspondence to an anomaly.

We show that our approaches not only allow discovery of multiple variable-length anomalous subsequences at once, but also significantly outperform the current state-of-the-art exact algorithms for time series anomaly detection.

1. INTRODUCTION

The ability to detect anomalies in time series efficiently is important in a variety of application domains where anomalies convey critical and actionable information, such as in health care, equipment safety, security surveillance, and fraud detection. Consequently, the anomaly detection problem has been studied in diverse research areas [10]. Despite the problem's simplicity at the abstract level, where an anomaly is defined as a pattern that does not conform to the underlying generative processes, the problem is difficult to solve in its most general form [3].

Anomalies in time series can be divided into two broad categories: point anomalies and structural anomalies. Point anomalies are statistical outliers, i.e., points which are significantly different from others [11], and have been studied the most [3]. In contrast, structural anomalies, whose discovery is our present focus, are defined as subsequences whose shape do not conform to the rest of the observed, or expected patterns [10, 3, 13].

Previously in [13], the notion of time series discord was introduced. Discords are shown to capture in a sense the most unusual subsequences within a time series that are likely to correspond to many possible anomalies within the generative processes – a property which was confirmed in a recent extensive empirical study by Chandola et al., where they concluded *"..on 19 different publicly available data sets, comparing 9 different techniques time series discord is the best overall technique among all techniques"* [3]. However, to discover a discord, the user must specify its length. There are two limitations with this requirement in real world problems. First, the user may not know the exact discord length, or even the best range of lengths in advance. Second, restricting the discovery to only fixed length discords limits the algorithm's exploratory capacity since multiple discords of different lengths may co-exist in a time series. As a result, determining all possible lengths to discover the best discords would be extremely cost prohibitive.

In this work, we focus on the discovery of structural anomalies that can also be described as *the most unusual subsequences within a given time series*, and we introduce a framework that addresses the above limitation by enabling efficient detection of variable-length anomalies. The proposed algorithms relies on the grammar induction procedure, which once applied to a string obtained by symbolic time series discretization, learns algorithmically exploitable symbol correlations and builds a hierarchical structure of context-free grammar rules, each of which maps to *variable-length* subsequences of the input time series. Through the

10.5441/002/edbt.2015.42

analysis of the grammar's hierarchical structure, the algorithms efficiently identify substrings that are rarely used in the grammar rules and whose corresponding subsequences can be considered as candidate anomalies.

Our approach builds upon the general notion of Kolmogorov complexity [15], which defines a string's complexity as a size of the smallest program that generates the string. While the Kolmogorov complexity is an uncomputable function due to the undecidability of the Turing machine halting problem, its value is typically approximated by the size of the input string in its algorithmically compressed form, and the tightness of the approximation bound is related to the overall efficiency of the compressor [27, 17]. This practical notion of *algorithmic compressibility* allows for the estimation, study, and application of Kolmogorov complexity in a number of generic solutions to common data mining tasks. For example it underlies the Minimum Description Length (MDL) [9] and Normalized Compression Distance (NCD) [5] principles, and has been used for time series anomaly discovery [14].

Within the algorithmic compressibility framework, the *algorithmic (Kolmogorov) randomness* of a string has been defined through its incompressibility, i.e., the lack of algorithmically exploitable redundancy [17, 9, 6, 20]. Since a grammar induction algorithm can be used to provide effective and efficient compression [21], naturally, it can be used for both the estimation of Kolmogorov complexity and algorithmic randomness discovery. Hence our present goal is to explore this property and to show that the algorithmic randomness discovered with the application of grammar induction-based compression to discretized time series can be correlated to the anomalousness within the time series.

In summary, our work has the following significant contributions:

- To the best of our knowledge, we are the first to explore the application of grammar-based compression to the problem of time series anomaly discovery.
- We propose two novel techniques for time series anomaly discovery based on grammatical compression, in which we define an anomaly as an incompressible, algorithmically random subsequence.
- Our approaches offer the unique ability to discover multiple variable-length anomalies at once (Figure 1).

The remainder of the paper is organized as follows. In Section 2, we provide notation and define our research problem. In Section 3, we give a motivational example and describe algorithms used. We discuss our approach in detail in Section 4, showing two algorithms enabling grammatical compression-driven anomaly detection in time series. In Section 5, we empirically evaluate our algorithms on datasets as diverse as spatial trajectories, space shuttle telemetry, medicine, surveillance, and industry. We also show the utilities of incorporating the algorithms into our visualization tool, GrammarViz 2. Finally, we review related work and conclude.

2. NOTATION AND THE PROBLEM DEFINITION

To precisely state the problem at hand, and to relate our work to previous research, we will define the key terms used throughout this paper. We begin by defining our data type, time series:

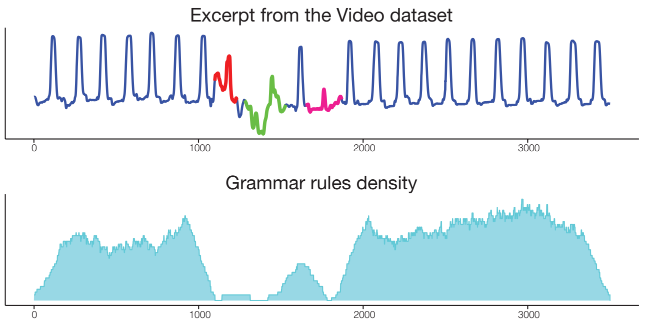


Figure 1: An example of multiple anomalous events found in a recorded video time series [14] shown at the top panel. The *rule density curve*, which we propose in this paper, and which is built in linear time and space, is shown in the bottom panel. Reflecting the hierarchical grammar structure, the rule density curve reaches its minima where no algorithmic redundancy is observed, pinpointing anomalous locations precisely.

Time series $T = t_1, \ldots, t_m$ is a set of scalar observations ordered by time.

Since we focus on the detection of anomalous patterns, which are likely to be local features, we consider short subsections of time series called subsequences:

Subsequence C of time series T is a contiguous sampling $t_p, \ldots, t_{p+n-1}$ of points of length $n << m$ where p is an arbitrary position, such that $1 \leq p \leq m - n + 1$.

Typically subsequences are extracted from a time series with the use of a sliding window:

Sliding window subsequence extraction: for a time series T of length m, and a user-defined subsequence length n, all possible subsequences of T can be found by sliding a window of size n across T.

As it is well acknowledged in the literature, and as we have shown before in [25], it is often meaningless to compare time series unless they are z-normalized:

Z-normalization is a process that brings the mean of a subsequence C to zero and its standard deviation to one.

Given two time series subsequences C and M, both of length n, the distance between them is a real number that accounts for how much these subsequences are different, and the function which outputs this number when given C and M is called the **distance function** and denoted $Dist(C, M)$. One of the most commonly used distance functions is the **Euclidean distance**, which is the square root of the sum of the squared differences between each pair of the corresponding data points in C and M.

One of our proposed techniques is built upon determining if a given subsequence C is similar to other subsequences M under distance measure $Dist$. This notion is formalized in the definition of a match:

Match: Given a positive real number t (i.e., threshold) and subsequences C and M, if $Dist(C, M) \leq t$ then subsequence M is a match to C.

When searching for potential anomalies using a distance function, it is important to exclude self matches, which are subsequences that overlap the subsequence currently being considered. Such self-matches can yield degenerate and unintuitive solutions as discussed in [13]. For two subsequences C and M we define a non-self match:

Non-self match: Given a subsequence C of length n starting at position p of time series T, the subsequence M beginning at q is a non-self match to C at distance $Dist(C, M)$ if $|p - q| \geq n$.

As mentioned, one of the most effective methods for time series anomaly detection is via discord discovery. Formally, it is defined as:

Time Series Discord: Given a time series T, the time series subsequence $C \in T$ is called the discord if it has the largest Euclidean distance to its nearest non-self match [13]. Thus, time series discord is a subsequence within a time series that is maximally different to all the rest of subsequences in the time series, and therefore naturaly captures the most unusual subsequence within the time series [13].

2.1 Problem definition

The task of finding a structural time series anomaly is defined as

Given a time series T, find a subsequence C that is the most (structurally) different from the rest of the observed subsequences.

This task, however, is very difficult to solve in its general form without a notion of the context [3]. The context is information that can be induced from the structure of the dataset or specified as a part of the problem. It places constraints on both the search space and the results, making it possible to find a meaningful solution. Based on this rationale, we re-define the anomaly discovery problem as:

Given a time series T and some context, find a subsequence C that is the most structurally different from others and which can be related to the context.

In discords—the current state of the art in structural anomaly detection [3]—the context is provided by the user-defined anomaly length, and the notion of the "most structurally different" is defined as the largest Euclidean distance to the nearest non-self match. Both constraints, while defining the problem and the solution exactly, place severe restrictions on the result by assuming unrealistic a priori knowledge about the exact anomaly length.

In this work we address this issue by allowing the discord length to vary in boundaries that are consistent with the time series context. Toward this end, we represent the context as a hierarchical grammar structure obtained through the processes of time series symbolic discretization and context-free grammar induction. In turn, by exploiting the use frequencies of the induced grammar rules, our technique finds the most unusual rules which we consider as discord candidates to be evaluated and which, naturally, vary in length.

3. GRAMMAR-BASED TIME SERIES DECOMPOSITION

Before describing our approach in detail, consider the following example showing the context-free grammar properties used in our approach. Let

$$S = abc \;\; abc \;\; cba \;\; xxx \;\; abc \;\; abc \;\; cba$$

be the input string under analysis (e.g. derived from a time series and reflecting its structure). For reason that will become clearer later, the input string consists of a sequence of *words* (in this example, 3-letter words or triplets). Each triplet is considered an atomic unit, or a *terminal* in the sequence. The task is to compress this input sequence by grammar induction.

A careful look at the string shows that there are repeated patterns *abc abc cba* separated by *xxx*. Ideally, we expect the grammar induction or compression algorithm to reflect this, as shown in the possible grammar for input S:

Grammar Rule	Expanded Grammar Rule
R0 → R1 xxx R1	abc abc cba xxx abc abc cba
R1 → R2 cba	abc abc cba
R2 → abc abc	abc abc

As shown, the grammar induction algorithm has reduced the length of the input string (i.e., compressed it) by creating a grammar whose rules are encoded by *non-terminals* $R1$ and $R2$, which reveal repeated patterns in the input.

In previous work, we have shown that by the analysis of a grammar built upon time series discretization it is possible to identify recurrent patterns, i.e. time series motifs [16]. Since anomaly detection can be viewed as the inverse problem to motif discovery, in this work, we argue that symbols that are rarely used in grammar rules (i.e. xxx) may aid in anomaly detection as well. The intuition is that subsequences of any length that never or rarely occur in grammar rules are non-repetitive and are thus most likely to be unusual or anomalous.

To illustrate this, suppose we annotate each word of the input string S with the number of rules that the word appears in excluding the top-level rule R0. The input string S becomes the following:

$$S = abc_2 \; abc_2 \; cba_1 \; xxx_0 \; abc_2 \; abc_2 \; cba_1$$

All occurrences of the word *abc* have a count of 2 because they appear in both $R1$ and $R2$; the word *cba* has count of 1 since it appears only in $R1$; whereas the word xxx has a count 0, because it is not a part of any rule. Since the counts naturally reflect the algorithmic compressibility of the sequence of terminal and non-terminal symbols, the triplet xxx_0 is algorithmically incompressible by the grammar induction algorithm and thus algorithmically random. In turn, if the input string S is derived by discretizing a time series into a sequence of words, where each word corresponds to a time series subsequence, then based on our hypothesis, the subsequence in the time series that xxx represents is a potential anomaly.

Note that when identifying a potential anomaly we have not used any explicit distance computation between terminal or non-terminal symbols, grammar rules, or their corresponding (i.e., raw) subsequences. Moreover, note that the time series discretization technique SAX [25] and the grammatical inference algorithm Sequitur [22] that we rely upon, also do not compute any distance (i.e., they do not explicitly measure how far apart objects are). Hence, unlike most anomaly discovery algorithms, our approach does not require any distance computation to discover and to rank multiple potential anomalies.

Discovered in the above example potential anomaly is the most unusual substring of a larger input string in terms of the grammatical inference algorithm of choice. Specifically, in contrast to other terminal symbols, the word xxx is not included in any of grammatical rules – the property that is discovered and accounted for by the grammatical inference algorithm. Thus, the discovered anomalous substring is analogous in meaning to a *time series discord*. However, our approach determines the anomalous subsequence length automatically in the course of grammar induction process, whereas the discord discovery algorithm requires the length of a potential anomaly to be known in advance.

Based on the intuition shown above, we shall present two algorithms that enable the discovery of variable-length anomalies. Before that, we discuss time series discretization and grammatical inference – the procedures upon which our techniques are built.

3.1 Discretization

Since grammar induction algorithms are designed for discrete data, we begin by discretizing a continuous time series with SAX (Symbolic Aggregate approXimation) [25]. In addition, since an anomaly is a local phenomenon, we apply SAX to subsequences extracted via a sliding window. SAX performs discretization by dividing z-normalized subsequence into w equal-sized segments. For each segment, it computes a mean value and maps it to symbols according to a pre-defined set of breakpoints dividing the distribution space into α equiprobable regions, where α is the alphabet size specified by the user. This *subsequence discretization* process [19] outputs an ordered set of SAX words, where each word corresponds to the leftmost point of the sliding window, and which we process with numerosity reduction at the next step.

As an example, consider the sequence $S1$ where each word (e.g. *aac*) represents a subsequence extracted from the original time series via a sliding window and discretized with SAX (the subscript following each word denotes the starting position of the corresponding subsequence in the time series):

$$S1 = aac_1\ aac_2\ abc_3\ abb_4\ acd_5\ aac_6\ aac_7\ aac_8\ abc_9\ \ldots$$

In contrast to many SAX-based anomaly discovery techniques that store SAX words in a trie or a hash table for optimizing the search, and essentially throw away the ordering information, we argue that the sequential ordering of SAX words provides valuable *contextual information*, and is the key for allowing variable-length pattern discovery.

3.2 Numerosity reduction

As we have shown in [19], neighboring subsequences extracted via sliding window are often similar to each other. When combined with the smoothing properties of SAX, this phenomenon persists through the discretization, resulting in a large number of consecutive SAX words that are identical. Later, these yield a large number of trivial matches significantly affecting performance. To address this issue, we employ a numerosity reduction strategy: if in the course of discretization, the same SAX word occurs more than once consecutively, instead of placing every instance into the resulting string, we record only its first occurrence. Applied to $S1$, this process yields:

$$S1 = aac_1\ \ abc_3\ \ abb_4\ \ acd_5\ \ aac_6\ \ abc_9$$

In addition to speeding up the algorithm and reducing its space requirements, the numerosity reduction procedure provides an important feature in this work – it naturally enables the discovery of variable-length anomalies as we show next.

3.3 Grammar induction on SAX words

Next, the reduced (from repetitions) sequence of SAX words is inputted into Sequitur [22], our grammar induction algorithm of choice, to build a context-free grammar.

Sequitur is a linear time and space algorithm that derives the context-free grammar from a string incrementally. Processing the input string from left to right, Sequitur builds the hierarchical structure of a context-free grammar by identifying and exploiting symbol correlations while maintaining the two constraints of uniqueness and utility at all times. Although simple in design, Sequitur has been shown to be competitive with state of the art compression algorithms – the property which allows us to use the notion of Kolmogorov complexity. In addition, Sequitur performance tends to improve with the growth of the input string size [21].

When applied to a sequence of SAX words, Sequitur treats each word as an input string token and builds the context-free grammar's hierarchical structure. This structure recursively reduces all *digrams* that are consecutive pairs of tokens (terminal or non-terminal) occurring more than once in the input string to a single new non-terminal symbol.

To reiterate the benefit of the numerosity reduction strategy and how it lends itself to variable-length pattern discovery with Sequitur, consider the single grammar rule $R1$ generated by Sequitur from the string *S1* as shown here:

Grammar Rule	Expanded Grammar Rule
R0 $\rightarrow$ R1 abb acd R1	aac_1 abc_3 abb_4 acd_5 aac_6 abc_9
R1 $\rightarrow$ aac abc	*aac abc*

In this grammar, $R1$ concurrently maps to substrings of different lengths: $S1_{[1:3]}$ of length 3 (i.e., $aac_1\,aac_2\,abc_3$) and $S1_{[6:9]}$ of length 4 (i.e., $aac_6\,aac_7\,aac_8\,abc_9$), respectively. The potential anomalous substring "$abb_4\,acd_5$" has length 2. Since each SAX word corresponds to a *single point* of the input time series (a subsequence starting point), $R1$ maps to its subsequences of variable lengths.

3.4 Mapping rules to subsequences

As shown in the above example, by keeping SAX words' offsets throughout the procedures of discretization and grammar induction, our algorithm is able to map rules and SAX words back to their original time series subsequences.

3.5 Pattern mining with Sequitur

Previously in [16], we proposed GrammarViz, an algorithm for variable-length time series motif discovery that makes full use of the hierarchy in Sequitur's grammar. We showed the ability of the proposed algorithm to discover recurrent patterns of variable lengths. This is due to several properties of the algorithm, including: the data smoothing capability of SAX, numerosity reduction which enables the patterns' variable length, and Sequitur's *utility* constraint which ensures that all of the grammar's non-terminals correspond to recurrent patterns. We later implemented visualization software based on this concept [26] that also provides a pilot module demonstrating the potential for a grammar-based approach to identify anomalies.

In this work, we formally introduce the notion of the *rule density curve* which is the key to our grammar-driven anomaly detection algorithm. Simply put, the rule density curve reflects the number of Sequitur grammar rules that span a time series point. We also provide theoretical background for our empirical observations. For this, we emphasize the role of the second Sequitur constraint, *digram uniqueness*, which ensures that none of the digrams processed by the algorithm (i.e., compressed into non-terminals) repeats itself. This property guarantees the *exhaustiveness of the search* for algorithmically exploitable redundancies in the input string, and consequently *asymptotically maximal compression* of the output string [21]. Both properties allow us to put our approach within the Kolmogorov complexity framework based on the algorithmic compressibility and relate algorithmically incompressible subsequences to anomalies as we discuss in the next section.

4. GRAMMAR-DRIVEN ANOMALY DISCOVERY

Within Kolmogorov complexity research, it has been proven that *algorithmic incompressibility is a necessary and sufficient condition for randomness* [6, 17], thanks to the elegant statistically-sound theory developed by Martin-Löf [20]. This theoretically grounds our intuition and effectively supports the claim that if a grammar induction algorithm is incapable of encoding a subsequence by finding exploitable correlations within the input string, such a subsequence is random within the context of the input string and applied algorithm. We call such subsequences *algorithmically anomalous* and equate them to time series anomalies.

Let us explain the utility of "algorithmic anomalousness". When searching for an anomaly in a time series, we expect that while the true generative process is unknown, it is likely to be regular and that the time series reflects these regularities. At the same time, we also assume that the time series may contain some abnormal segments, whose identification is our goal. Further, assuming that the discretization process preserves these regularities and irregularities, the Sequitur algorithm should be able to learn the regularities and effectively compress the input string. However, due to its invariants of utility and uniqueness, Sequitur will not be able to form rules that contain symbolic subsequences occurring just once in the input string, because it will not be able to find any short- or long-term correlations between them and the rest of the string – the property that reflects irregularity and defines a variable-length anomaly in the most natural way.

Based on the intuition behind algorithmic anomalousness we propose two algorithms for grammatical compression-driven variable-length anomaly discovery from time series. Configured only by the discretization parameters, both algorithms are capable of efficient discovery of putative anomalous subsequences without any prior knowledge of their length, shape, or minimal occurrence frequency. While the result produced by the first algorithm is an approximate solution, our second algorithm is based on explicit distance computations and outputs time series discords of variable length.

4.1 Efficient, rule density-based anomaly discovery

To efficiently discover approximate anomalies, we propose to compute the rule density curve for the input time series. Toward that end, an empty array of length m (the length of the time series), is first created. Each element in this array corresponds to a time series point and is used to keep count of the grammar rules that span (or "cover") the point. Second, since the locations of corresponding subsequences for all grammar rules are known, by iterating over all grammar rules the algorithm increments a counter for each of the time series points that the rule spans. After this process

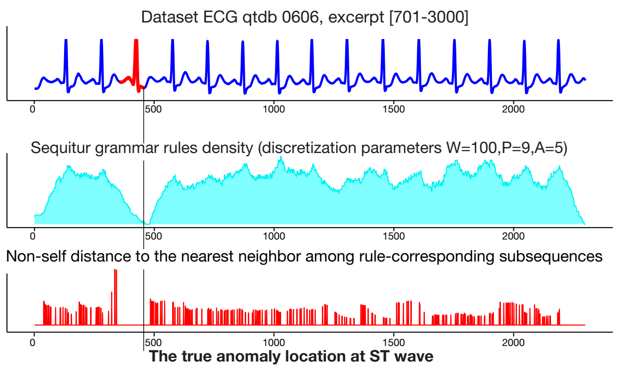


Figure 2: Anomaly discovery in ECG dataset. Top panel shows the anomalous heartbeat location. Middle panel shows that the rule density curve clearly identifies the true anomaly by its global minimum. Bottom panel confirms that the RRA-reported discord has indeed the largest distance to its nearest non-self match.

each element of the array contains a value indicating the total number of grammar rules that covers the corresponding time series point. The curve that corresponds to the array's values is the *rule density curve.* As an example, consider the rule density curves shown in the middle panels of Figures 2 and 3.

Since each SAX string corresponds to a subsequence starting at some position of the time series, the points whose rule density counters are global minima correspond to the grammar symbols (terminals or non-terminals) whose inclusion in the grammar rules are minimal. These subsequences are algorithmically anomalous by our definition and we argue that the *rule density curve intervals that contain minimal values correspond to time series anomalies*, and our algorithm simply outputs these intervals.

Consider the example shown in Figure 2. The top panel shows an excerpt of an ECG time series with a highlighted instance of an anomalous heartbeat featuring a very subtle premature ventricular contraction. The middle panel shows a significant drop in the grammar rule density over the interval 462-484, which is in perfect alignment with the ground truth – an expert's annotation of an anomaly occuring in the ST interval of the ECG curve (as discussed in [13]). Similar to that, the global minima of the rule density curve shown in the middle panel of Figure 3 pinpoints the weekly interval that has the most unusual power consumption pattern (the dataset in the top panel of Figure 3 shows the power consumption history of a Dutch research facility for the entire year of 1997 [28]).

The rule density-based approach is capable of discovering multiple anomalies of variable length. When given a fixed threshold, it simply reports contiguous points of the input time series whose density is less than the threshold value. If needed, an additional ranking criterion can be defined, such as a minimal anomaly length or a statistically sound criterion based on probabilities.

Note that even though we need to specify the sliding window length, it is only the initial "seed" value. Unlike most existing algorithms in which this subsequence length is the exact length of the anomaly, anomalies reported by our technique are not bounded by the seed length and may range from very short to very long time spans.

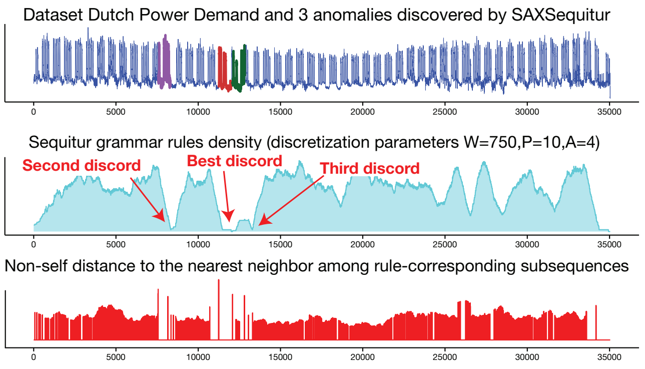


Figure 3: Multiple discord discovery in Dutch power demand data [28]. Top panel shows 52 weeks of power demand by a research facility. Middle panel shows that while the rule density-based technique was able to discover the best discord, others are difficult to discriminate. The bottom panel shows distances to the nearest non-self match computed for each rule-corresponding subsequence, which allows for the ranking of discords discovered with RRA.

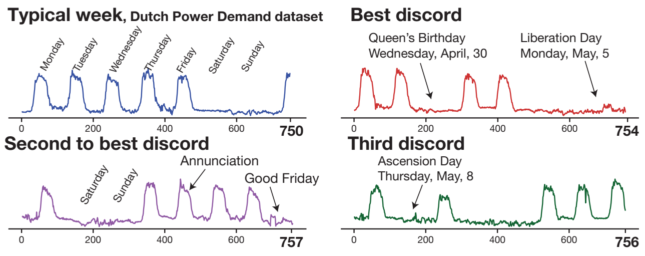


Figure 4: A detailed view of RRA-ranked variable length discords discovered in the Dutch power demand dataset. All of them highlight time intervals where typical weekly patterns are interrupted by state holidays.

Another distinguishable and desirable characteristic of this approach is its efficiency. It has linear time and space complexity since the sequential processing of SAX, Sequitur, and the global minima search take linear time and space. This efficiency, when combined with effective rule density curve-based visualization, enables the user to interactively explore the dataset and to refine discretization parameters and the anomaly selection threshold.

4.2 Exact, distance-based anomaly discovery

If the time series under analysis has low regularity (an issue that impacts the grammar's hierarchy) or the discretization parameters are far from optimal and regularities are not conveyed into the discretized space, the rule density-based anomaly discovery technique may fail to output true anomalies. In addition, some applications may require additional anomaly evidence or ranking. To address this, we propose a second variant of a grammar-driven variable-length anomaly discovery algorithm based on an explicit distance computation which outputs discords – the subsequences whose distance to their nearest non-self match is the largest. Since anomalous subsequences correspond to rare grammar rules, we call the algorithm RRA (Rare Rule Anomaly).

Algorithm 1 RRA algorithm

```
 1: function FIND_DISCORD($T, Intervals, Outer, Inner$)
 2:     best_so_far_dist = 0
 3:     best_so_far_loc = NaN
 4:     for each $p$ in $Intervals$ ordered by $Outer$ do
 5:         nearest_neighbor_dist = Infinity
 6:         for each $q$ in $Intervals$ ordered by $Inner$ do
 7:             if $|p_0 - q_0| \geq Length(p)$ [1] then
 8:                 current_dist = $Dist(p, q)$
 9:                 if current_dist < best_so_far_dist then
10:                     break
11:                 if current_dist < nearest_neighbor_dist then
12:                     nearest_neighbor_dist = current_dist
13:         if nearest_neighbor_dist > best_so_far_dist then
14:             best_so_far_dist = nearest_neighbor_dist
15:             best_so_far_loc = p
16:     return (best_so_far_dist, best_so_far_loc)
```

[1]p_0 and q_0 are the global indexes (in T) of the first points of subsequences p and q respectively. In this line we check that currently analyzed subsequences do not overlap (i.e., q is non-self match of p).

The RRA algorithm is based on the HOTSAX framework initially proposed in [13] for the discord discovery. The algorithm's input includes the original time series T, a list of variable length subsequences corresponding to grammar rules which we call $Intervals$, and two heuristics: $Inner$ and $Outer$, which can be applied to list of subsequences.

Similar to HOTSAX, our algorithm iterates over all candidate subsequences in the outer loop (line 4 of Algorithm 1) while computing distances to all other non-self matches in the inner loop (lines 6–8; p_0 and q_0 in line 7 are the indexes) and selecting the closest non-self match (lines 9–15). The candidate subsequence from the outer loop which yields the largest distance to a non-self match is output as the result. In HOTSAX, the candidates in the outer ($Outer$) and inner ($Inner$) loops are ordered based on the SAX representations of the candidate subsequences such that the order of consideration is as close to the *optimal* ordering (i.e., the ordering that would result in the most elimination of computations) as possible. However, as mentioned earlier, HOTSAX candidates are restricted by their subsequence length. Our proposed technique differs from HOTSAX in that subsequences (i.e., $Intervals$ in Algorithm 1) and their ordering for the inner ($Inner$) and outer ($Outer$) loops are provided as the input based on the information derived from grammar.

Specifically, $Intervals$ subsequences are those that correspond to the grammar rules plus all continuous subsequences of the discretized time series that do not form any rule. The $Outer$ subsequence ordering utilizes the information derived from a hierarchical grammar structure – we order subsequences in ascending order of their corresponding rule usage frequency (note that continuous subsequences of the discretized time series that do not form any rule have frequency 0 and are thus considered first). The intuition behind this ordering is simple and is a reflection of the previously discussed properties of algorithmically anomalous subsequences. That is, the sooner we encounter the *true* anomaly, the larger the $best\_so\_far\_dist$ is, and the more computations we can potentially eliminate later on (line 9).

The $Inner$ candidate match ordering is also based on grammar information. First, having a candidate subsequence

Table 1: Performance comparison for brute-force, state-of-the-art, and the proposed exact discord discovery algorithms.

Dataset name and discretization param. (window, PAA, alphabet)	Length	Number of calls to the distance function			Reduction in distance calls	*HOTSAX* & *RRA* discords length and overlap	
		Brute-force	*HOTSAX*	*RRA*			
Daily commute (350,15,4)	17'175	271'442'101	879'067	112'405	87.2%	350 / 366	100.0%
Dutch power demand (750,6,3)	35'040	1.13×10^9	6'196'356	327'950	95.7%	750 / 773	96.3%
ECG 0606 (120,4,4)	2'300	4'241'541	72'390	16'717	76.9%	120 / 127	79.2%
ECG 308 (300,4,4)	5'400	23'044'801	327'454	14'655	95.5%	300 / 317	97.7%
ECG 15 (300,4,4)	15'000	207'374'401	1'434'665	111'348	92.2%	300 / 306	65.0 %
ECG 108 (300,4,4)	21'600	441'021'001	6'041'145	150'184	97.5%	300 / 324	89.7%
ECG 300 (300,4,4)[i]	536'976	288×10^9	101'427'254	17'712'845	82.6%	300 / 312	83.0%
ECG 318 (300,4,4)	586'086	343×10^9	45'513'790	10'000'632	78.0%	300 / 312	80.7%
Respiration, NPRS 43 (128,5,4)	4'000	14'021'281	89'570	45'352	49.3%	128 / 135	96.0%
Respiration, NPRS 44 (128,5,4)	24'125	569'753'031	1'146'145	257'529	77.5%	128 / 141	61.7%
Video dataset (gun) (150,5,3)	11'251	119'935'353	758'456	69'910	90.8%	150 / 163	89.3%
Shuttle telemetry, TEK14 (128,4,4)	5'000	22'510'281	691'194	48'226	93.0%	128 / 161	72.7%
Shuttle telemetry, TEK16 (128,4,4)	5'000	22'491'306	61'682	15'573	74.8%	128 / 138	65.6%
Shuttle telemetry, TEK17 (128,4,4)	5'000	22'491'306	164'225	78'211	52.4%	128 / 148	100.0%

[i] RRA reported the best discord discovered with HOTSAX as the second discord (Figure 5).

from a grammar rule selected in the *Outer* loop, we consider all other subsequences from the same rule as possible nearest non-self matches. After this step, the rest of the subsequences are visited in random order. The intuition behind this ordering is also simple – the subsequences corresponding to the same Sequitur rule are very likely to be highly similar. Thus, considering those in the beginning of *Inner* loop allows us to potentially encounter a distance that is smaller than *best_so_far_dist* sooner and to benefit from early abandoning (lines 9–10 of the Algorithm 1) while considering all other candidates in the *Outer* loop. Since RRA operates with rule-corresponding subsequences of variable lengths, when searching for nearest non-self match we employ the Euclidean distance normalized by the subsequence length, which favors shorter subsequences for the same distance value:

$$Dist(p,q) = \frac{\sqrt{\sum_{i=1}^{n}(p_i - q_i)^2}}{Length(p)} \qquad (1)$$

When run iteratively, excluding the current best discord from *Intervals* list, RRA outputs a ranked list of multiple co-existing discords of variable length, as shown in Figures 3 and 4. The bottom panels of Figures 2 and 3 indicate locations and true distances from each time series subsequence corresponding to a grammar rule to its nearest non-self match by a vertical line placed at the rule beginning and whose height equals the distance.

5. EXPERIMENTAL EVALUATION

We evaluated both proposed techniques on a number of datasets previously studied in [13] that include Space Shuttle Marotta Valve telemetry (TEK), surveillance (Video dataset), health care (electrocardiogram and respiration change), and industry (Dutch Power Demand). We also evaluated on a new dataset of spatial trajectories. We compared the performance of the proposed algorithms against brute force and HOTSAX [13] discord discovery algorithms. Since RRA returns discords of variable length that may differ significantly from the specified sliding window length, we show the RRA discord recall rate as the overlap between discords discovered by HOTSAX and RRA algorithms in the last column of Table 1.

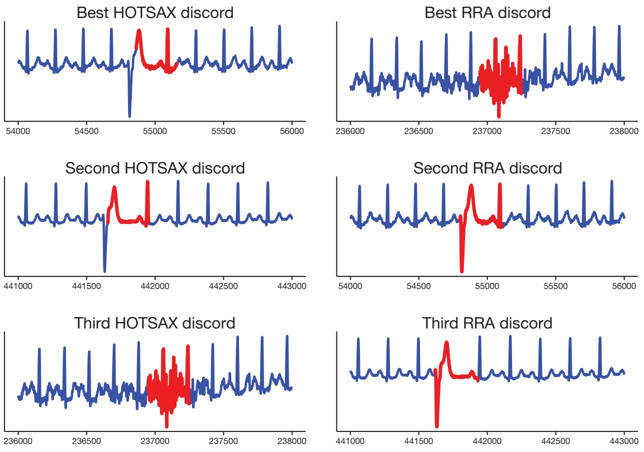


Figure 5: The comparison of discords ranking by HOTSAX and RRA algorithms from ECG300 dataset of length 536'976. RRA ranked the shorter discord first due to the larger value of normalized by the subsequence length Euclidean distance (Eq.(1)) to its nearest non-self match: the best discord has length 302, whereas the second and third discords have a length of 312 and 317 respectively.

We compared the algorithms performance in terms of calls to the distance computation routine, which, as pointed out in [13], typically accounts for up to 99% of these algorithms' computation time. Table 1 compares the number of distance function calls made by the competing techniques. Note that in the ECG300 dataset (which is record 300 of the MIT-BIH ST change database [8]), RRA failed to rank discords in the same order as the HOTSAX algorithm.

Our rule density-based algorithm was also able to discover anomalies in **all** data sets, though more careful parameter selection was needed at times; nevertheless, we found that this technique allows the discovery of very short anomalies which other evaluated techniques missed. For example, in the spatial trajectory dataset, the rule density-based technique was the only method capable of discovering a short, true anomaly that was intentionally planted by taking a detour.

To summarize, the rule density-based approach, when used alone, is extremely fast, but it has difficulty discriminating

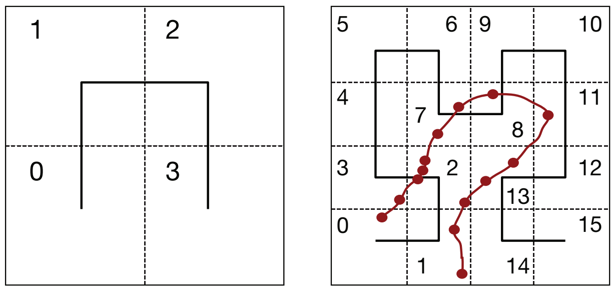


Figure 6: Approximations of the Hilbert space filling curve (first order at the left, second order at the right panel) and a trajectory conversion example. The trajectory shown at the right panel is converted into the sequence {0,3,2,2,2,7,7,8,11,13,13,2,1,1} by converting each recorded spatial position into the enclosing Hilbert cell id.

and ranking subtle discords. Incorporating the grammatical context into the distance-based RRA algorithm, however, enables the efficient discovery of discords in all data sets. RRA is much faster than HOTSAX and brute force, and it allows for the discovery of variable-length discords.

5.1 Spatial trajectory case study

To demonstrate the utility of our technique for discovering anomalies of an *unknown nature*, we performed a case study on spatial trajectory data. The trajectory data is intrinsically complex to explore for regularity since patterns of movement are often driven by unperceived goals and constrained by unknown environmental settings.

The data used in this study was gathered from a GPS device which recorded location coordinates and times while commuting during a typical week by car and bicycle.

To apply RRA to the trajectory, the multi-dimensional trajectory data (time, latitude, longitude) was transformed into a sequence of scalars. To achieve this, the trajectory points were mapped to the visit order of a Hilbert space filling curve (SFC) [12] embedded in the trajectory manifold space and indexed by the recorded times in the visit order (Figure 6, right panel). The Hilbert SFC was chosen to reduce the distortion on the data's spatial locality. The Hilbert SFC-transformed trajectory produces a time series, which is then passed to the RRA algorithm for anomaly discovery.

To visualize this data transformation approach, consider Figure 6 showing a Hilbert SFC of first order in the left panel and one of second order in the right panel. Note, that the left panel is divided into 4 quadrants and the first-order curve is drawn through their center points. The quadrants are ordered such that any two which are adjacent in the ordering share a common edge. In the next step, shown in the right panel, each of the quadrants of the left panel are divided into 4 more quadrants and, in all, 4 "scaled-down" first order curves are drawn and connected together. Note that the adjacency property of consecutive squares is maintained. As shown, maintaining adjacency helps to preserve spatial locality – points close in space are generally close in their Hilbert values. For our trajectory experimentation, we have used a Hilbert SFC of order eight.

In general, a trajectory anomaly is defined as a sub-trajectory path that is atypical in the set of paths taken by an individual. Specifically, an anomaly can either be a sub-

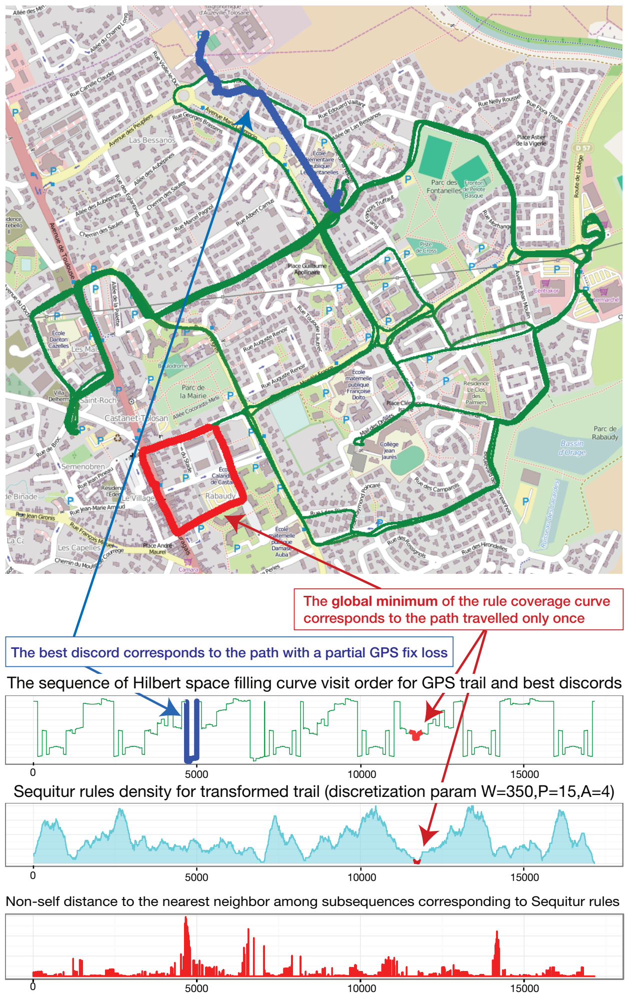

The global minimum of the rule coverage curve corresponds to the path travelled only once
The best discord corresponds to the path with a partial GPS fix loss
The sequence of Hilbert space filling curve visit order for GPS trail and best discords
0
5000
10000
15000
Sequitur rules density for transformed trail (discretization param W=350,P=15,A=4)
Non-self distance to the nearest neighbor among subsequences corresponding to Sequitur rules

Figure 7: An example of anomaly discovery in the Hilbert SFC transformed GPS track. The true anomaly, corresponding to the unique detour, was discovered by the rule density curve global minima, which reaches 0 at the interval of length 9, the best RRA discord of length 366 corresponds to the path traveled with a partial GPS fix (abnormal path running across properties). Note that RRA approach was not able to capture the anomalous detour.

trajectory that occurs in rarely visited spatial regions such as a detour, or a novel path taken within a frequently visited spatial region. The second type of trajectory anomaly is important because it considers the order in which the various locations are visited. For instance, if multiple points in a space are visited frequently, the occurrence of a visit to these points is not an anomaly by itself; however, the occurrence of visiting these points in an unseen order is an anomaly. To evaluate the proposed algorithm's efficiency in these specific settings, we also intentionally planted an anomaly by taking an atypical route.

Figure 7 shows the results of the discovered anomalies in the GPS track by both proposed algorithms. As shown, the rule density curve pinpoints an unusual detour deviating from a normal route significantly (red colored segment), the RRA algorithm highlighted a trajectory segment which was travelled with a partial GPS signal fix, but close to previously traveled routes (blue segment). This results highlight the difference in the algorithms' sensitivity due to

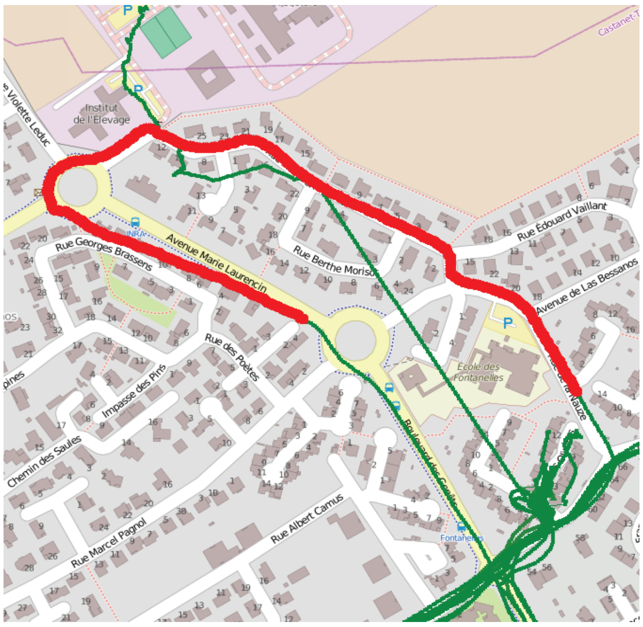


Figure 8: The second discord discovered by the RRA algorithm highlights a uniquely traveled segment.

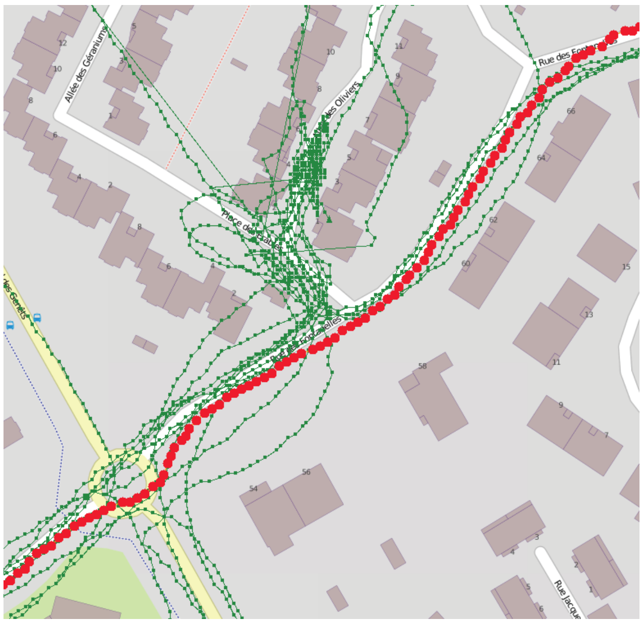


Figure 9: The third discord discovered by the RRA algorithm highlights an abnormal behavior that does not conform to the usual pattern of exiting and entering the block's parking lot.

their nature: the rule density curve-based approach finds algorithmically anomalous, short subsequences (shorter than the specified sliding window length) in the symbolic space of discretized values, whereas RRA is capable to rank algorithmically similar symbolic subsequences by discordance using their real representation.

While the second RRA-discovered discord shown in Figure 8 highlights a unique path, the third discord shown in Figure 9 spotlights the algorithm's sensitivity and ability to

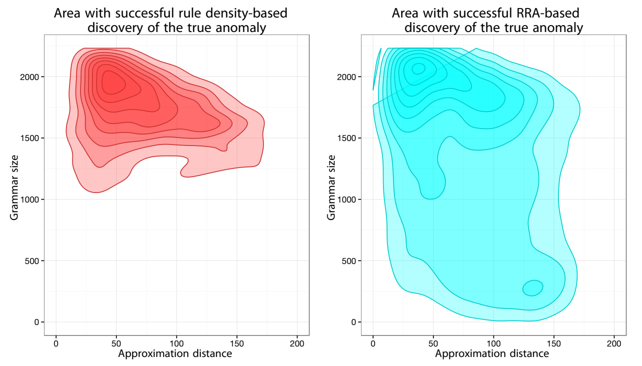


Figure 10: An illustration from our exploratory study concerned with optimal parameters selection based on the constructed grammar properties. Both plots show boundaries of optimal parameter choices: left panel for the rule density curve-based algorithm, right panel for RRA. Note that RRA algorithm-corresponding area is much larger indicating its robustness.

capture subtle anomalies. The shown discord corresponds to an abnormal behavior within frequently traveled spatial regions – not visiting the block's parking lot when traveling through the area.

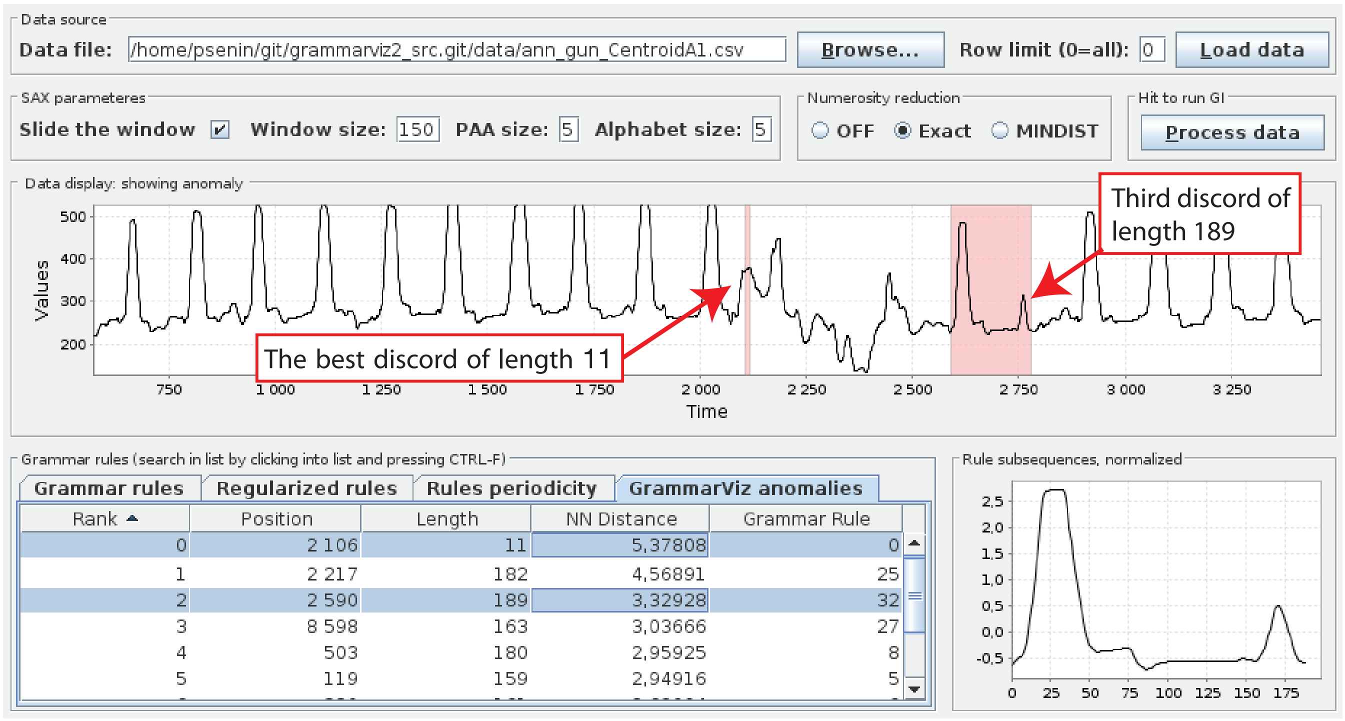
5.2 Discretization parameters selection

Similar to other discretization-based learning techniques, it is difficult to pinpoint a solution that offers the best tradeoff between gain in tractability and loss in accuracy. Nevertheless, we found that within the grammar-based paradigm, the sliding window length parameter is not as critical as it is for most of the existing anomaly and motif detection algorithms, since it is just the "seed" size. Specifically, we found that the rule density curve facilitates the discovery of patterns that are much shorter than the window size, whereas the RRA algorithm naturally enables the discovery of longer patterns. Second, we observed that when the selection of discretization parameters is driven by the context, such as using the length of a heartbeat in ECG data, a weekly duration in power consumption data, or an observed phenomenon cycle length in telemetry, sensible results are usually produced.

In addition, we found that the rule density approach alone is more sensitive to parameter choices than it is when incorporated into the RRA distance-based algorithm. Consider an example shown in Figure 10 where we used the ECG0606 dataset featuring a single true anomaly (the dataset overview is shown in Figure 2). Since the discretization parameters affect both the precision of raw signal approximation and the size of the resulting grammar, by sampling this space and recording both algorithms' results we found that the area where the RRA algorithm discovered the true anomaly is twice as large as the same area for the rule density curve-based algorithm. In particular, when we varied the window size in the range $[10, 500]$, PAA size in $[3, 20]$, and the alphabet size in $[3, 12]$; the rule density curve-based algorithm successfully discovered the anomaly for 1460 parameter combinations whereas RRA for 7100.

5.3 Visualization

As we pointed out before, to explore the properties of algorithmically anomalous subsequences, we have incorporated both algorithms proposed in this paper into our visualiza-


Data source
Browse...
Load data
SAX parameteres
Numerosity reduction
Hit to run GI
Process data
Data display: showing anomaly
Values
Time
Third discord of length 189
The best discord of length 11
Grammar rules (search in list by clicking into list and pressing CTRL-F)
Grammar rules
Regularized rules
Rules periodicity
GrammarViz anomalies
Rule subsequences, normalized

Data file: /home/psenin/git/grammarviz2_src.git/data/ann_gun_CentroidA1.csv

Row limit (0=all): 0

Slide the window ✓ Window size: 150 PAA size: 5 Alphabet size: 5

○ OFF [✓ Exact] ○ MINDIST

Rank	Position	Length	NN Distance	Grammar Rule
0	2 106	11	5,37808	0
1	2 217	182	4,56891	25
2	2 590	189	3,32928	32
3	8 598	163	3,03666	27
4	503	180	2,95925	8
5	119	159	2,94916	5

Figure 11: Incorporating the RRA algorithm in GrammarViz 2.0 [26]. The screenshot shows its application to the recorded video data set [14]. As shown, when configured with a window length of 150, RRA was able to detect multiple discords whose lengths vary from 11 to 189.

tion tool called GrammarViz 2.0 [26]. Figure 11 shows the screenshot of GrammarViz 2.0 using the RRA algorithm to find anomalies in a recorded video dataset. The discovered candidate anomalies are ranked by the distances to their nearest non-self matches. As shown in the "Length" column, all candidate anomalies have different lengths. The highlighted subsequences in the upper panel correspond to the anomalies selected in the bottom panel.

Figure 12 shows the anomalies discovered in the same dataset using the rule density curve-based approach. As shown, we use the blue color intensity to express the grammar rules density: the darker is the shade, the higher is the corresponding value in rule density curve (i.e., the higher is rule count). Thus, the white-shaded regions denote the best potential anomalies since they correspond to global minima intervals in the rule density curve.

Incorporating the proposed algorithms in our visualization tool allows interactive and efficient user-driven parameter tuning, as well as navigation and visualization of the results.

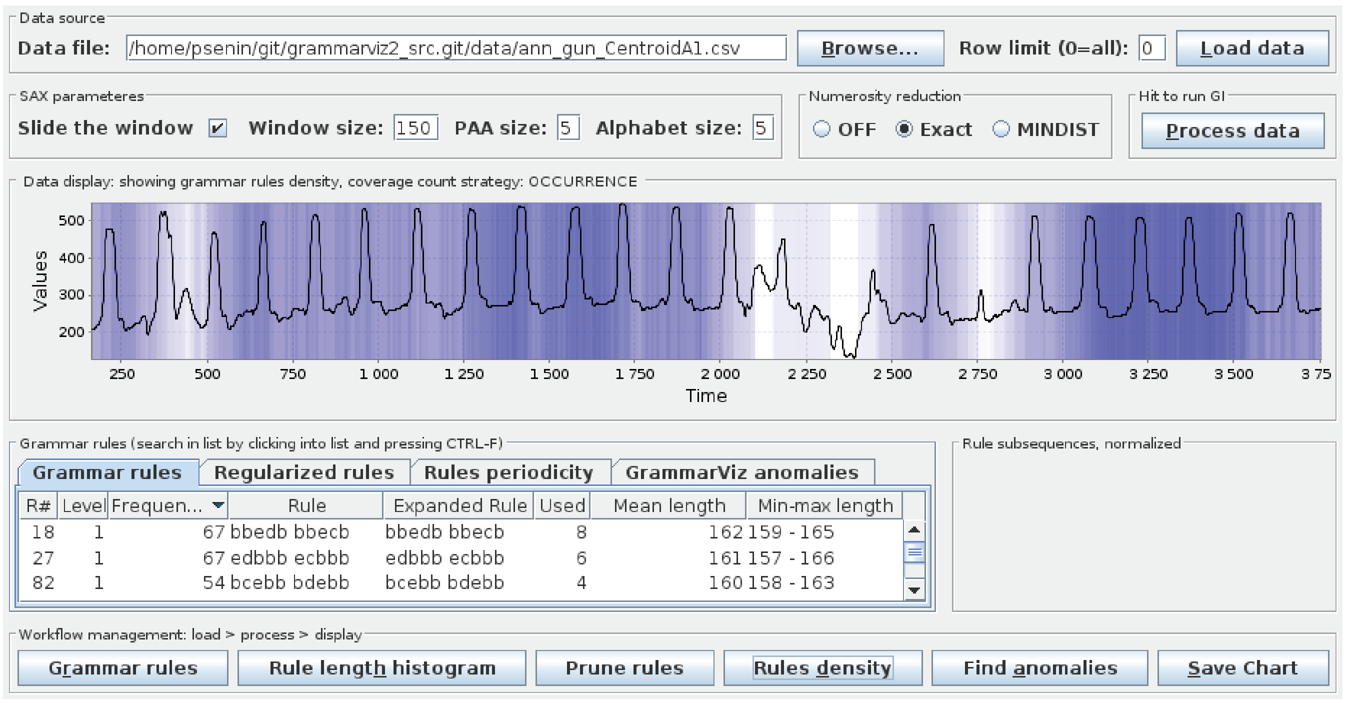
6. PREVIOUS WORK ON ANOMALY DETECTION

The brute force solution for the problem of time series anomaly detection or, more specifically, the discovery of a discord of a given length n in time series T of length m, needs to consider all possible distances between each subsequence C of length n and all of its non-self matches M $(C, M \in T)$. This method has $O(m^2)$ complexity and is simply untenable for large data sets.

To mitigate this heavy computational requirement, previous work suggests that the subsequence comparisons should be reordered for efficient pruning. For example HOTSAX [13], which is the pioneering work on discord discovery, suggests a fast heuristic technique that is capable of true discord discovery by reordering subsequences by their potential degree of discordance. Similarly in [29], the authors use locality sensitive hashing to estimate similarity between shapes with which they can efficiently reorder the search to discover unusual shapes. The authors of [7] and [2] use Haar wavelets and augmented tries to achieve effective pruning of the search space. While these approaches achieve a speed-up of several orders of magnitude over the brute-force algorithm, their common drawback is that they all need the length of a potential anomaly to be specified as the input, and they output discords of a fixed length. In addition, even with pruning, they rely on the distance computation which, as suggested by Keogh et al. [13], accounts for more than 99% of these algorithms run-time.

An interesting approach to find anomalies in a very large database (terabyte-sized data set) was shown by Yankov et al. [31]. The authors proposed an algorithm that requires only two scans through the database. However, this method needs an anomaly defining range r as the input. In addition, when used to detect an unusual subsequence within a time series, it also requires the length of the potential discord.

Some techniques introduced approximate solutions that do not require distance computation on the raw time series. VizTree [18] is a time series visualization tool that allows for the discovery of both frequent and rare (anomalous) patterns simultaneously. VizTree utilizes a trie (a tree-like data structure that allows for a constant time look-up) to decode the frequency of occurrences for all patterns in their discretized form. Similar to that defined in VizTree, Chen et


Data source

Data file: /home/psenin/git/grammarviz2_src.git/data/ann_gun_CentroidA1.csv

Row limit (0=all): 0

SAX parameteres

Slide the window ☑ Window size: 150 PAA size: 5 Alphabet size: 5

Numerosity reduction

○ OFF [✓ ● Exact] ○ MINDIST

Data display: showing grammar rules density, coverage count strategy: OCCURRENCE

Grammar rules (search in list by clicking into list and pressing CTRL-F)

R#	Level	Frequen...	Rule	Expanded Rule	Used	Mean length	Min-max length
18	1	67	bbedb bbecb	bbedb bbecb	8	162	159 - 165
27	1	67	edbbb ecbbb	edbbb ecbbb	6	161	157 - 166
82	1	54	bcebb bdebb	bcebb bdebb	4	160	158 - 163

Rule subsequences, normalized

Workflow management: load > process > display

Figure 12: Incorporating the rule-density-curve approach in GrammarViz 2.0 [26]. The varying degrees of shades in the background correspond to *rule density* curve values; the non-shaded (white) intervals pinpoint true anomalies.

al.[4] also consider anomalies to be the most infrequent time series patterns. The authors use support count to compute the anomaly score of each pattern. Although the definition of anomalies by Chen et al. is similar to discords, their technique requires more input parameters such as the precision of the slope e, the number of anomalous patterns k, or the minimum threshold. In addition, the anomalies discussed in their paper contain only two points. Wei et al. [30] suggest another method that uses time series bitmaps to measure similarity.

Finally, some previous work has examined the use of algorithmic randomness for time series anomaly discovery. Arning et al. [1] proposed a linear time complexity algorithm for the sequential data anomaly detection problem from databases. Simulating a natural mechanism of memorizing previously seen data entities with regular-expression based abstractions capturing observed redundancy, their technique has been shown capable of detecting deviations in linear time. The proposed method relies on the user-defined entity size (the database record size). Alternatively, Keogh et al. [14] have shown an algorithmic randomness-based parameter-free approach to approximate anomaly detection (the WCAD algorithm). However, built upon use of an off-shelf compressor, their technique requires its numerous executions, which renders it computationally expensive; in addition, it requires the sliding window (i.e., anomaly) size to be specified.

7. CONCLUSION AND FUTURE WORK

In this work we hypothesized, that time series anomaly maps to algorithmically anomalous (i.e., incompressible with a grammatical inference algorithm) symbolic subsequence within the string obtained via time series symbolic discretization. The rationale behind this hypothesis is that if true, it allows for an efficient variable-length time series anomaly discovery approach.

Building upon subsequence discretization with SAX, which preserves the data structural context, and grammar induction with Sequitur, which guarantees discovery of all existing algorithmically-effective correlations by maintaining its invariants of uniqueness and utility at all times, we designed a generic framework for learning algorithmic regularities and detecting irregularities in time series in order to test our hypothesis.

Using the framework, we constructed two time series anomaly discovery algorithms to empirically evaluate the hypothesis. One of these algorithms operates in the space of discretized data, whose dimensionality is typically much smaller than the original time series, and therefore is highly efficient. The output of this algorithm, namely the *rule density curve*, was found to behave according to our hypothesis and provides an intuitive and efficient way for putative anomaly detection. Our second algorithm is based on the explicit distance computation and is capable to detect even subtle *variable-length discords.*

Through an experimental evaluation, we have validated our hypothesis and have shown, that the proposed techniques are orders of magnitude more efficient than current state of the art without a loss in accuracy (Table 1).

Since the grammar-based time series decomposition allows us to quantitatively assess the time series context through analysis of the grammar's hierarchical structure, the primary direction of our future effort is to analyze the effect of the discretization parameters on the algorithm's ability to discover contextually meaningful patterns. Since both techniques underlying our approach, namely, SAX discretization and grammatical inference with Sequitur, process the input time series from left to right, yet another research direction that suggests itself is the possibility of early anomaly detection in real-time data streams.

8. ACKNOWLEDGMENTS

This research is partially supported by the National Science Foundation under Grant No. 1218325 and 1218318.

9. REFERENCES

[1] Arning, A., Agrawal, R., & Raghavan, P., *A Linear Method for Deviation Detection in Large Databases,* In KDD (pp. 164-169) (1996)

[2] Bu, Y., Leung, O., Fu, A., Keogh, E., Pei, J., Meshkin, S., *WAT: Finding Top-K Discords in Time Series Database*, In Proc. of SIAM Intl. Conf. on Data Mining (2007)

[3] Chandola, V., Cheboli, D., and Kumar, V., *Detecting Anomalies in a Time Series Database*, CS TR 09–004 (2009)

[4] Chen, X., Zhan, Y., *Multi-scale Anomaly Detection Algorithm based on Infrequent Pattern of Time Series*, J. of Computational and Applied Mathematics (2008)

[5] Cilibrasi, R., Vitányi, P.M.B., *Clustering by compression*, IEEE Trans. Inform. Theory (2005)

[6] Ferbus-Zanda, M., Grigorieff, S., *Is Randomness "Native" to Computer Science?*, arXiv (2008)

[7] Fu, A., Leung, O., Keogh, E., Lin, J., *Finding Time Series Discords based on Haar Transform*, In Proc. of Intl. Conf. on Adv. Data Mining and Applications (2006)

[8] Goldberger, A.L. et al., *PhysioBank, PhysioToolkit, and PhysioNet: components of a new research resource for complex physiologic signals*, Circulation, 101(23) (2000)

[9] Grünwald, P. D., *The Minimum Description Length Principle*, MIT Press (2007)

[10] Gupta, M., Gao, J., Aggarwal, C.C., Han, J., *Outlier Detection for Temporal Data: A Survey*, IEEE Trans. on Knowledge and Data Engineering, 25, 1 (2013)

[11] Hawkins, D. M., *Identification of Outliers*, Chapman and Hall (1980)

[12] Hilbert, D., *Üeber stetige Abbildung einer Linie auf ein Flächenstück*, Mathematische Annalen, 38:459–460 (1891)

[13] Keogh, E., Lin, J., Fu, A., *HOT SAX: Efficiently Finding the Most Unusual Time Series Subsequence*, In Proc. ICDM'05 (2005)

[14] Keogh, E., Lonardi, S., Ratanamahatana, C.A., *Towards parameter-free data mining*, In Proc. KDD (2004)

[15] Kolmogorov, A.N., *Three approaches to the quantitative definition of information*, Problems Inform. Transms. (1965)

[16] Li, Y., Lin, J., and Oates, T., *Visualizing variable-length time series motifs*, In Proc. of SDM (2012)

[17] Li, M. and Vitányi, P.M.B., *An Introduction to Kolmogorov Complexity and Its Applications*, Springer-Verlag (1993)

[18] Lin, J., Keogh, E., Lonardi, S., Lankford, J.P., Nystrom, D. M., *Visually mining and monitoring massive time series*, In Proc. ACM SIGKDD Intn'l Conf. on KDD (2004)

[19] Lin, J., Keogh, E., Patel, P., and Lonardi, S., *Finding Motifs in Time Series*, The 2nd Workshop on Temporal Data Mining, the 8th ACM Int'l Conference on KDD (2002)

[20] Martin-Löf, Per, *On the definition of random sequences*, Information and Control, MIT, 9:602-61 (1966)

[21] Nevill-Manning, C. and Witten, I., *Linear-Time, Incremental Hierarchy Inference for Compression*, In Proc. of IEEE Conference on Data Compression (1997)

[22] Nevill-Manning, C. and Witten, I., *Identifying Hierarchical Structure in Sequences: A linear-time algorithm*, Journal of Artificial Intelligence Research, 7, 67-84 (1997)

[23] Oates, T., Boedihardjo, A., Lin, J., Chen, C., Frankenstein, S., Gandhi, S., *Motif discovery in spatial trajectories using grammar inference*, In Proc. of ACM CIKM (2013)

[24] Paper authors. Supporting webpage: http://github.com/GrammarViz2 (2015)

[25] Patel, P., Keogh, E., Lin, J., Lonardi, S., *Mining Motifs in Massive Time Series Databases*, In Proc. ICDM (2002)

[26] Senin, P., Lin, J., Wang, X., Oates, T., Gandhi, S., Boedihardjo, A.P., Chen, C., Frankenstein, S., Lerner, M., *GrammarViz 2.0: a tool for grammar-based pattern discovery in time series*, In Proc. ECML/PKDD (2014)

[27] Solomonoff, R.J., *A formal theory of inductive inference. Part I*, Rockford Research Institute, Inc. (1962)

[28] Van Wijk, J.J. and Van Selow, E.R., *Cluster and calendar based visualization of time series data*, In Proc. IEEE Symposium on Information Visualization (1999)

[29] Wei, L., Keogh, E., Xi, X., *SAXually explicit images: Finding unusual shapes*, In Proc. ICDM (2006)

[30] Wei, L., Kumar, N., Lolla, V., Keogh, E., Lonardi, S., Ratanamahatana, C., *Assumption-free Anomaly Detection in Time Series*, In Proc. SSDBM (2005)

[31] Yankov, D., Keogh, E., Rebbapragada, U., *Disk aware discord discovery: finding unusual time series in terabyte sized data sets*, Knowledge and Information Systems, 241-262 (2008)

K-Nearest Neighbor Temporal Aggregate Queries

Yu Sun †1, Jianzhong Qi †2, Yu Zheng ‡3, Rui Zhang †4

† *Department of Computing and Information Systems, University of Melbourne, Victoria, Australia*

{1 sun.y, 2 jianzhong.qi, 4 rui.zhang}@unimelb.edu.au

‡ *Microsoft Research, Beijing, P.R.China*

3 yuzheng@microsoft.com

ABSTRACT

We study a new type of queries called the *k-nearest neighbor temporal aggregate* (kNNTA) query. Given a query point and a time interval, it returns the top-k locations that have the smallest weighted sums of (i) the spatial distance to the query point and (ii) a temporal aggregate on a certain attribute over the time interval. For example, *find a nearby club that has the largest number of people visiting in the last hour*. This type of queries has emerging applications in location-based social networks, location-based mobile advertising and social event recommendation. It is a great challenge to efficiently answer the query due to the highly dynamic nature and the large volume of the data and queries. To address this challenge, we propose an index named TAR-tree, which organizes locations by integrating the spatial and temporal aggregate information. We perform a detailed analysis on the cost of processing kNNTA queries using the TAR-tree. The analysis shows that the TAR-tree results in much fewer node accesses than alternatives. Furthermore, we propose two enhancements for the kNNTA query: (i) an algorithm suggesting the least amount of weights to be adjusted to explore different query results and (ii) a collective processing scheme to share index traversal among a batch of queries. We conduct extensive experiments using real-world data sets. The results validate the accuracy of the cost analysis and show that the TAR-tree outperforms alternatives by up to ten times in node accesses. The results also show that the weight adjustment algorithm and collective processing scheme outperform their baselines by significant margins.

1. INTRODUCTION

Location-based services (LBSs) have a large market and this market is growing rapidly. A well-known global market research company MarketsandMarkets forecasts in a recent report that the LBSs market will grow from $8.12 billion in 2014 to $39.87 billion in 2019. Location-based social networks (LBSNs) [31] have been a driving force for the growth of LBSs. Many emerging applications enable users to explore their neighborhood with rich social information in a highly customized fashion. For example, using the functionality *Places Nearby* (e.g., in Facebook or Foursquare), users may want to find nearby attractions that have the most visits recently or find a nearby club that is gathering the most people in the last hour; using the functionality *Explore* (e.g., in Flickr or Instagram), users may want to browse photos taken nearby and have the most *likes* lately.

These applications require ranking locations (or geotagged media contents) based on two criteria: (i) the spatial distance and (ii) a temporal aggregate on a certain attribute (e.g., the visits or likes). The spatial distance indicates the degree of closeness while the temporal aggregate reflects the social opinion in a certain period. These applications exhibit three key characteristics, which create a highly dynamic environment: (i) The visits or likes happen continuously, making the aggregate data grow rapidly. For instance, there were 3 million check-ins per day in Foursquare by May 2014. The number of the aforementioned requests is also very large. (ii) The time interval a user interested in is highly customized, which may vary from hours (e.g., for retrieving current events) to years (e.g., for long term analyses). (iii) The users may adjust their weighting on the two criteria widely to explore results of different preferences.

The skyline operator [6] can support multi-criteria decision problems. However, the skyline operator is computationally expensive even for static data and queries. The highly dynamic environment and the large volume of requests and objects generated in LBSNs make it prohibitive to use the skyline operator. Moreover, users are not given the flexibility in determining their preference over the two criteria. Following existing studies [9][15][22], we rank the locations using a weighted sum of the spatial distance and the temporal aggregate. We formulate the problem as the *k-nearest neighbor temporal aggregate* (kNNTA) query (formally defined in Section 3). Apart from the above applications in LBSNs, kNNTA queries are useful in many other applications in urban computing [32] where the spatial distance and a temporal aggregate are considered simultaneously, such as location-based mobile advertising and social event recommendation.

The kNNTA query requires quick response since users usually use the query to browse locations or geotagged media contents in the neighborhood. Due to the dynamic nature and the huge volume of the data and queries, having an efficient solution to this type of queries is challenging. Existing indexing structures cannot manage the locations effectively

10.5441/002/edbt.2015.43

based on both spatial closeness and temporal aggregate information simultaneously (detailed discussion in the related work, Section 2). To efficiently process the kNNTA query, we propose a novel index named the TAR-tree, in which the locations are organized by integrating the spatial and temporal aggregate information. We perform a detailed analysis on the cost of query processing using the TAR-tree. The analysis shows that the TAR-tree results in much fewer node accesses than alternatives that organize the locations based on only the spatial or the temporal aggregate information. The analysis can also be used as a cost model for query optimization. Furthermore, we propose two enhancements for the kNNTA query: (i) To help users explore results of different preferences, we propose an efficient algorithm suggesting the least amount of weights to be adjusted between the two criteria so that the query results will change. (ii) To handle large number of queries, we propose a collective processing scheme to share index traversal among a batch of queries. In summary, the main contributions of this paper are as follows.

- We propose a query called the k-nearest neighbor temporal aggregate (kNNTA) query to address emerging applications that requires ranking locations on both (i) the spatial distance and (ii) a temporal aggregate on a certain attribute.
- We propose a novel index named the TAR-tree to efficiently process the kNNTA query. We perform a detailed analysis on the cost of query processing using the TAR-tree, which shows that the TAR-tree results in much fewer node accesses than alternatives.
- We propose two enhancements for the kNNTA query: (i) an algorithm suggesting the least amount of weights to be adjusted to explore different query results and (ii) a collective processing scheme to share index traversal among a batch of queries.
- We conduct extensive experiments using real-world data sets. The results validate the accuracy of the cost analysis, and show that the TAR-tree outperforms alternatives by up to ten times in node accesses. The results also show that the weight adjustment algorithm and collective processing scheme outperform their baselines by significant margins.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 formalizes the kNNTA query. Section 4 presents the TAR-tree. Section 5 discusses grouping strategies. Section 6 provides the analysis. Section 7 gives two enhancements. Section 8 reports the experiment results and Section 9 concludes the paper.

2. RELATED WORK

Queries. Previous spatial aggregate queries focus on the *range aggregate* [25], which returns the summarized information of POIs falling in a hyper rectangle (e.g., find the maximum or minimum weight among POIs intersecting the query rectangle). Temporal range aggregate queries [26] have also been studied, which add the temporal dimension to range aggregate queries (e.g., return the number of cars in the city center *during the last hour*). The kNNTA query differs from these queries in that (i) it returns the POIs rather than the aggregate value (e.g., the number of cars) and (ii) its aggregate is over the history of individual POIs (e.g., the check-in history) rather than spatial regions. Spatial keyword queries [9] retrieve the top-k objects such that their locations are close to the query point and their textual descriptions are relevant to the query keywords. The kNNTA query differs from spatial keyword queries in that instead of the keywords query time intervals are given, and a dynamic aggregate attribute (e.g., the count of check-ins) rather than the textual relevance is considered. Given two data sets P and Q (queries), an aggregate nearest neighbor (aNN) query [19] retrieves the points in P that have the smallest aggregate distances to the points in Q. The aNN query aggregates on the distances of a group of points, different from the kNNTA query which aggregates in the time dimension. Therefore, the algorithms for aNN queries cannot apply. Many other types of queries aggregating on different objects such as moving objects [10][16], data streams [29] or locations [13][20][21] are also studied. These queries are all different from the kNNTA query, and hence the algorithms for them cannot apply.

Indexes. Indexes such as aR-tree [17] and aP-tree [25] were proposed to process range aggregate queries. They cannot be adapted to process the kNNTA query because only one aggregate is maintained. The kNNTA query requires the temporal aggregate over various time intervals. Papadias et al. [26] proposed the aRB-tree to process temporal range aggregate queries. The aRB-tree combines the R-tree and B-tree, making each entry of the R-tree point to a B-tree which stores historical aggregates of the entry over each timestamp. To address the *distinct counting problem* in aRB-tree, i.e., an object will be counted multiple times if it remains in the query rectangle for more than one timestamp, Tao et al. [24] proposed the *sketch index* which is similar to the aRB-tree but with the B-tree storing historical counting sketches of the regions in its subtree. The aRB-tree and sketch index cannot be adapted to process the kNNTA query when the epochs are of varied lengths, since the B-tree cannot index time intervals. Even if the epochs are of equi-length, the aRB-tree and sketch index pay no attention to entry grouping strategies and group the entries based on only spatial extents, which, as will be shown in our analysis and experiments, is not effective for processing the kNNTA query. Sun et al. [23] divided the space into regular grid and proposed an adaptive multi-dimensional histogram (AMH) to answer temporal range aggregate queries. AMH cannot be adapted to answer the kNNTA query either, since the histogram buckets only maintain the aggregate and cannot retrieve individual POIs. Even if we use extremely fine granularity such that each cell in the grid only contains one POI, the buckets are grouped mainly by the aggregate dimension which, as will be shown, is also an ineffective strategy. Cong et al. [9] proposed the IR-tree for spatial keyword queries by integrating the R-tree and inverted indexes. Variants of the IR-tree, such as DIR-tree, group the R-tree entries by minimizing a weighted sum of the spatial closeness and text similarities, which is not optimal since it introduces another parameter, precludes existing optimization techniques for R-tree and makes it difficult to estimate the query processing cost. When designing the TAR-tree, our main focus is to develop a robust and effective grouping strategy. Many other spatial indexes [14][30] for nearest neighbor queries are also proposed. These indexes cannot be adapted to process the kNNTA query as they only focus on the spatial dimensions and are unable to tackle the temporal aggregate.

3. PROBLEM FORMULATION

3.1 Query Definition

The locations, which may have spatial extents, are hereafter termed as points-of-interest (POIs). The visits, likes, and so on are termed as *checked-ins*. A *k-nearest neighbor temporal aggregate* (kNNTA) query returns the top-k POIs based on a weighted sum of (i) the spatial distance to the query point and (ii) a temporal aggregate on the check-ins over a time interval. More precisely, we rank the POIs by a function f that computes the ranking score of a POI p as

$$f(p) = \alpha_0 d(p, q) + \alpha_1(1 - g(p, \mathcal{I}_q)), \tag{1}$$

where $\alpha_i > 0$ (a constant) is the weight, $0 \le d(p, q) \le 1$ is the normalized Euclidean distance between p and the query point q, and $0 \le g(p, \mathcal{I}_q) \le 1$ is the normalized temporal aggregate of p over a query time interval $\mathcal{I}_q$. We use the weighted sum due to its simplicity and common usage in the literature [9][15][22], although the same result can be achieved by any monotonic function on the two criteria. We normalize the spatial distance $d(p, q)$ and temporal aggregate $g(p, \mathcal{I}_q)$ by dividing each by its *range* (i.e., maximum − minimum), so that the value is in the range $[0, 1]$. The normalization prevents one criterion from overpowering the other if it has a relatively large value. Without loss of generality, we let $\alpha_0 + \alpha_1 = 1$, since the ranking does not change if α_0 and α_1 is multiplied by a positive constant. The smaller the ranking score is, the higher p ranks and the better it suits the query.

The temporal aggregate can be *count*, *min*, *max*, *sum* or *average* (i.e., $\frac{sum}{count}$). In this paper, we focus on the aggregate that counts the number of check-ins at a POI, but the methods easily extend to other aggregates. In the rest of this paper, we omit "temporal" when the context is clear and simply use "aggregate" to refer to the "temporal aggregate". Let t_0 be the starting of the application and t_c be the current time. We discretize the time axis in to *epochs*. Each epoch may be a second, an hour or of varied lengths (e.g., one hour, two hours, four hours, eight hours and so on) depending on the application. The aggregate $g(p, \mathcal{I}_q)$ is computed by adding up the number of check-ins at p whose epoch intersects $\mathcal{I}_q$. We summarize the definition of the kNNTA query as follows.

DEFINITION 1. **K-Nearest Neighbor Temporal Aggregate (kNNTA) Query.** *Given a query point and a time interval, a k-nearest neighbor temporal aggregate query returns a set $\mathcal{R}$ of k POIs with the minimum ranking scores computed by the ranking function f given by Equation 1, i.e., $\forall p \in \mathcal{R}$ and $p' \in \mathcal{P} \setminus \mathcal{R}$, $f(p) \le f(p')$.*

3.2 A Straightforward Approach

Figure 1 gives an example. The circles are the POIs. Table 1 presents the number of check-ins that each POI has in epochs $[t_0, t_1)$, $[t_1, t_2)$ and $[t_2, t_c]$, respectively. A kNNTA query is issued with a query point q denoted by the small square, a time interval $[t_0, t_c]$, $\alpha_0 = 0.3$ ($\alpha_1 = 0.7$) and $k = 1$. The ranking score of e is computed by $f(\mathsf{e}) = 0.3 \cdot \frac{2.24}{15.6} + (1 - 0.3) \cdot (1 - \frac{2}{12}) = 0.626$, where 2.24 is the Euclidean distance between e and q, 15.6 is the maximum distance between any two points in the space, 2 is the aggregate at e over $[t_0, t_c]$ and 12 is the maximum aggregate among all POIs. We obtain f as the query result, whose spatial distance to q equals 3 and aggregate equals $3+5+4 = 12$. The ranking score of f is $0.3 \cdot \frac{3}{15.6} + (1-0.3) \cdot (1 - \frac{12}{12}) = 0.058$.

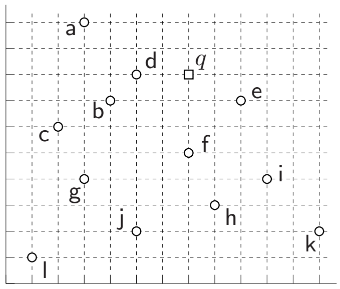


Figure 1: POIs and the query point

POI	$t_0\rightarrow$	$t_1\rightarrow$	$t_2\rightarrow$
a	1	1	0
b	1	0	1
c	2	2	2
d	2	0	0
e	1	1	0
f	3	5	4
g	2	3	1
h	1	1	0
i	2	2	2
j	2	0	0
k	1	0	1
l	1	0	1

Table 1: Aggregate distribution

To handle the kNNTA query, a straightforward approach is sequential scan. Assume that the check-ins have already been counted within each epoch (as shown in Table 1). We first add up the number of check-ins in each epoch in the query time interval and obtain the aggregate for each POI. We then compute the ranking score of each POI, and return the top-k POIs. The time complexity is $\mathcal{O}(m'\mathcal{N} + \mathcal{N} \log m + k \log \mathcal{N})$, where m' is the number of epochs in the query time interval, $\mathcal{N}$ is the number of POIs, m is the number of epochs in $[t_0, t_c]$ (e.g., 3 in the above example) and k is the number of returned POIs. Both $\mathcal{N}$ and m' are very large in real social networks. For instance, $\mathcal{N} = 60,000,000$ in the LBSN Foursquare, and $m' = 8,760$ if the query time interval is one year and each epoch is one hour. The high cost makes this approach inapplicable in real applications.

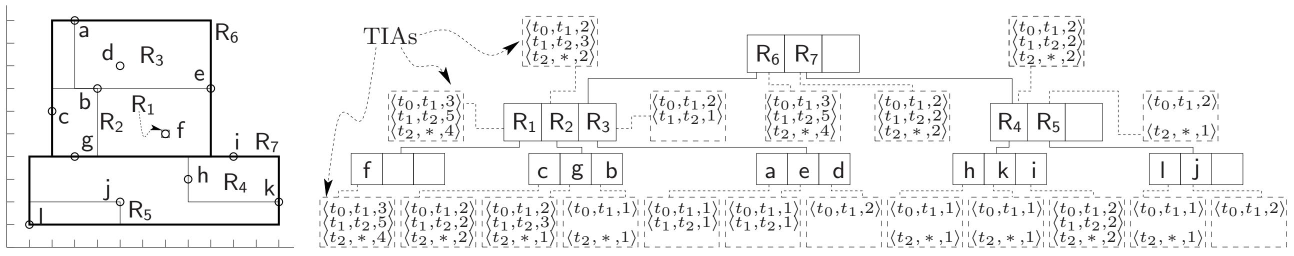
4. INDEX DESIGN

We design an index called the *temporal aggregate R-tree* (TAR-tree) to efficiently process the kNNTA query.

4.1 Index Structure

The TAR-tree is a variant of the R-tree.The algorithms for indexing the spatial extents of the POIs remain the same. A leaf entry is a minimum bounding rectangle (MBR) enclosing a POI. A leaf node contains a number of leaf entries. An entry in an internal node points to a child node (leaf node or internal node), and has an MBR enclosing the MBRs contained in the child node.

The difference between the TAR-tree and R-tree is that each entry of the TAR-tree also points to a temporal index. The temporal index stores the non-zero aggregate (at least one check-in) over each epoch, and keeps each record as a triple $\langle t_s, t_e, agg\rangle$, where t_s is the start time and t_e is the end time of the epoch, and *agg* is the aggregate value during the epoch. For brevity, we refer to the temporal index as the TIA (temporal index on the aggregate). The TIA of a leaf entry stores the aggregate of the POI it contains. The TIA of an internal entry stores the *largest* aggregate value of the TIAs in the child node for each epoch. For example, if two TIAs are in the child node and they store records $\{\langle t_0, t_1, 2\rangle, \langle t_1, t_2, 2\rangle, \langle t_2, *, 2\rangle\}$ and $\{\langle t_0, t_1, 2\rangle, \langle t_1, t_2, 3\rangle, \langle t_2, *, 1\rangle\}$, respectively, then the TIA of the internal entry pointing to this node stores the records $\{\langle t_0, t_1, \max\{2, 2\}\rangle, \langle t_1, t_2, \max\{2, 3\}\rangle, \langle t_2, *, \max\{2, 1\}\rangle\}$. Any temporal index can be used to implement the TIA. We have used the disk-based multi-version B-tree [2] in our implementation as it has been proven to be



(a) (b)

Figure 2: TAR-tree example

asymptotically optimal.

In most applications, the aggregate update (i.e., inserting check-ins) is much more frequent than the spatial update (i.e., inserting POIs). We maintain the spatial and aggregate information in different components to enable quick digestion of new check-ins. Figure 2 presents an example of TAR-tree indexing the POIs shown in Figure 1. Figure 2(a) shows the MBRs of the entries. Figure 2(b) shows the index structure. The temporal records indexed by TIAs are enclosed by dashed lines. Empty lines in the TIAs mean that no records are stored for the epoch due to a zero aggregate. As we will see, the most important aspect for TAR-tree to efficiently process the kNNTA query is the strategy to group the entries. We will discuss the entry grouping strategy in Section 5.

4.2 Index Maintance

We briefly discuss how to insert check-ins and POIs. Deletion is the same as R-tree and hence omitted.

Inserting Check-ins. When an epoch ends, we compute the aggregate of each POI by the check-ins (in this epoch), and then insert the non-zero aggregates in a batch fashion. Specifically, starting from the root node of TAR-tree, if an entry contains a POI whose aggregate is non-zero, we traverse the sub-tree rooted at the entry recursively. When reaching a leaf node, we store the non-zero aggregate into the POI's TIA, and return the largest aggregate in this node to the parent. Such an update procedure is efficient, since we only traverse part of the R-tree (which can be kept in main-memory) and insert only one record into the TIA.

Inserting POIs. When we insert a POI, the inserted path in TAR-tree is determined by the entry grouping strategy (which will be discussed in Section 5). For each entry in the inserted path, we update its MBR to include the POI, and update its TIA if in an epoch the aggregate of the POI is larger. If the insertion causes some POIs to be reinserted, we first remove these POIs from the TAR-tree, update the MBRs and TIAs in the inserted path, and then insert these POIs as described above. If the insertion causes some node to split, we redistribute the entries in the node by the entry grouping strategy.

4.3 Query Processing

We use the *best-first* search (BFS) [12] for query processing, which works as follows: (i) the entries in the root node are first inserted into a priority queue, in which the priority is determined by the entry's ranking score (detailed in the next paragraph), and then (ii) the front entry of the queue is ejected. If the entry is a leaf entry, the POI it contains is added to the result list; otherwise, each of its child entries is inserted into the queue. (iii) Step (ii) is repeated until k POIs are obtained.

The ranking score of an entry e is the weighted sum of the spatial distance from the query point to the MBR of e and the aggregate computed by the TIA of e. Given a query time interval $\mathcal{I}_q$, the TIA returns the records whose time interval $[t_s, t_e]$ is contained in $\mathcal{I}_q$. We obtain the aggregate over $\mathcal{I}_q$ by adding up the *agg* field of each returned record.

According to [12], the BFS produces correct query results as long as the entry's priority is computed by a *consistent* function. For the TAR-tree, the consistence can be expressed as: if e_c is an entry in the node pointed by entry e, then $f(e) \le f(e_c)$. We prove the consistency of the ranking function f as follows.

PROPERTY 1. *Given any query point q and query time interval $\mathcal{I}_q$, we have $f(e) \le f(e_c)$, where e_c is a child entry of entry e in the TAR-tree.*

PROOF. We have $f(e) = \alpha_0 d(e,q) + \alpha_1(1 - g(e, \mathcal{I}_q))$. Due to the TAR-tree design, it follows that $d(e,q) \le \min_{e_c \in e} d(e_c, q)$ and $g(e, \mathcal{I}_q) \ge \max_{e_c \in e} g(e_c, \mathcal{I}_q)$. Therefore,

$$f(e) \le \alpha_0 \min_{e_c \in e} d(e_c, q) + \alpha_1(1 - \max_{e_c \in e} g(e_c, \mathcal{I}_q))$$

$$\le \alpha_0 d(e_c, q) + \alpha_1(1 - g(e_c, \mathcal{I}_q)) = f(e_c) \ \forall e_c \in e,$$

i.e., $f(e) \le f(e_c)$. $\square$

5. ENTRY GROUPING STRATEGIES

We now discuss the strategies for grouping the TAR-tree entries. As proved above, the BFS will provide the correct query results on the TAR-tree no matter which grouping strategy is used. The BFS has been proven to be optimal per TAR-tree instance in that only the TAR-tree nodes that intersect the *search region* will be accessed by the BFS [4]. However, different entry grouping strategies may result in different TAR-tree instances and hence vastly different number of node accesses. The performance of the BFS on the TAR-tree is roughly proportional to the number of accessed nodes, since similar operations are performed on each accessed node and the TAR-tree is most likely disk resident due to its large size as we discussed in Section 4.1. Therefore, we aim at minimizing the *node extents* in the TAR-tree so that fewer nodes are accessed by the BFS.

5.1 Two Straightforward Strategies

Since the TAR-tree is a variant of the R-tree, one straightforward strategy is to group the entries based on the spatial extents as R-tree does. Here we briefly review the grouping method of R*-tree [3]. When inserting a POI, we choose the entry that has the least overlap with other entries after containing the POI, if the entry points to a leaf node. If the entry points to an internal node, we choose the one that has the least area enlargement after including the POI. When a node incurs overflow and this is the first time overflow happens in this level, we reinsert several entries of the node. When a node splits, we first choose a split axis, along which the sum of all possible new MBR margins is minimized. We then redistribute the entries (along the chosen split axis) such that the two new nodes have the minimum overlap.

Another straightforward strategy is to group the entries that have similar aggregate distributions. The similarity or distance between two aggregate distributions can be measured by the Manhattan distance (or Earth mover's distance and the like). For example, in Table 1, the distance between the TIA of c and TIA of g equals $0+1+1=2$, while the distance between the TIA of c and TIA of l equals $1+2+1=4$. When a POI is added, we insert the POI into the node that has the smallest distance to it. When a node splits, we redistribute the entries such that the distance between the two new nodes is maximized.

5.2 Integral 3D Strategy

As our analysis and experiments will show, the above two entry grouping strategies are not effective. We propose to group the entries by integrating the spatial and aggregate information to minimize the node extents. Specifically, we group the entries as 3-dimensional bounding boxes, in which two are the spatial dimensions and the third is a dimension capturing the aggregate information. As the aggregate information is distributed as aggregate values in many epochs. Here the trick lies in how to sufficiently represent the aggregate information as a single value (i.e., the coordinate of the third dimension). We have designed the third dimension as the following value

$$\widehat{\lambda}_p = \frac{1}{m}\sum_{i=1}^{m} v_i,$$

where m is the number of epochs in $[t_0, t_c]$ and v_i is the aggregate value in the i^{th} epoch (and as usual the bounding box of an internal entry encloses the bounding boxes of its child entries). This value is an estimate of the expected number of check-ins at the POI p contained by the leaf entry in an epoch (because we can model the number of check-ins at a POI in an epoch using the Poisson distribution). If two entries have similar such values, they may also have similar aggregates over the query time interval. It can significantly reduce the node extents if we group the entries having both similar spatial distances to the query point and similar aggregates over the query time interval.

Since the two types of information are of very different nature and do not have a unified domain range, when using this strategy, we normalize the spatial and aggregate dimensions by the ranges of their domains, respectively. In particular, to align with the ranking function, the normalized coordinate z_p of the third dimension for a leaf entry equals $z_p = 1 - \frac{\widehat{\lambda}_p}{\max_p \widehat{\lambda}_p}$. Note that only when we group the entries they are treated as 3-dimensional bounding boxes. When processing the kNNTA query, the spatial extents of the entry are obtained from the MBR and the aggregate from the TIA.

6. COST ANALYSIS AND COMPARISON OF GROUPING STRATEGIES

In this section, we analyze the query processing cost using the TAR-tree (with our proposed integral 3D entry grouping strategy). Through the cost analysis, we show that the TAR-tree results in much fewer node accesses than alternatives that use the other two grouping strategies. The analysis can also be used as a cost model for query optimization purposes. As mentioned before, we measure the cost by the number of node accesses. In the BFS, the accessed nodes are those intersecting the query search region, which is in turn determined by the data distribution. Therefore, we first analyze the distribution of the aggregate data in Section 6.1, and then estimate the search region and the number of node accesses in Sections 6.2 and 6.3, respectively. We compare the three entry grouping strategies in Section 6.4.

Table 2: Powerlaw fitting

Data	n	$\hat{\beta}$	$\hat{x}_{min}$	p-value
NYC	72,273	3.20	31	0.68
LA	45,591	3.07	16	0.18
GW	1,280,969	2.82	85	0.29
GS	182,968	2.19	59	0.21

6.1 Distribution of the Aggregate Data

Like many other types of data in real life [8], we observe that the aggregate value (i.e., the number of POIs having a certain aggregate value) follows the power-law distribution very well. Let the discrete random variable X be the count aggregate over a certain time interval, among the aggregates of all POIs, the probability that X has an observed value x is computed by

$$p(x) = \Pr(X = x) = Cx^{-\beta},$$

where C is a normalization constant. The power-law indicates that a small number of the POIs having a large proportion of the check-ins (roughly 80% of the check-ins are at 20% of the POIs). We test the power-law hypothesis on four real LBSN data sets (detailed at the beginning of Section 8) with the method in [8]. We list in Table 2 the results from the fitting of a power-law to each of the data sets, where n is the number of the tested POIs, $\hat{\beta}$ is the estimated scaling parameter, $\hat{x}_{min}$ is the estimated lower-bound to the power-law behavior and p-value is the goodness-of-fit indicator. It is suggested in [8] that the power-law hypothesis is ruled out if p-value is less than or equal to 0.1. Since the p-values of the four data sets are all clearly larger than 0.1, we argue that they all follow the power-law very well.

6.2 Estimation of the Query Search Region

Similar to the k-nearest neighbor query, the search region of the kNNTA query is determined by the ranking score of the k^{th} POI, which is denoted by $f(p_k)$. For ease of exposition, we describe the ranking score and search region in a normalized 3-dimensional unit cube, where two are the spatial dimensions and the third is the aggregate dimension. Figure 3 illustrates the ranking score with the query example in Section 3.2. The line segment $\overline{q\mathsf{g}'}$ represents the normalized spatial distance and $\overline{\mathsf{gg}'}$ represents the normalized aggregate of g. The ranking score of g equals $\alpha_0|\overline{q\mathsf{g}'}| + \alpha_1|\overline{\mathsf{gg}'}|$.

In the 3-dimensional unit cube, the query search region is of a cone shape. Its height and base radius, denoted by h_l and r_0, are computed by

$$r_0 = \frac{f(p_k)}{\alpha_0} \quad \text{and} \quad h_l = \frac{f(p_k)}{\alpha_1},$$

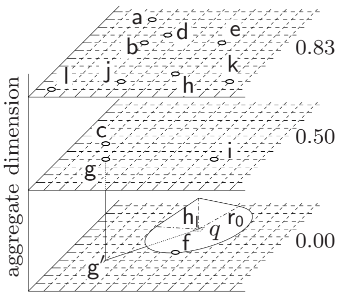


Figure 3: Cost analysis example

respectively. For example, in Figure 3, the cone illustrates the search region. Recall that in the query example we have $\alpha_0 = 0.3$, $\alpha_1 = 0.7$ and $f(p_k) = 0.058$, which implies that $r_0 = 0.192$ and $h_l = 0.082$. By definition, k POIs are in the search region. For instance, in the above example $k = 1$ and only f is in the search region. If $k = 2$, the search region will expand until it reaches a second POI. We use this property to estimate the size of the search region.

We observe that in the 3-dimensional unit cube, the POIs are only on a few *layers* at a specific height. Moreover, the number of such layers is countable. This is because the aggregate values (before normalization) are integers representing the number of check-ins. For example, in Figure 3, the POIs are only on three layers: a, b, d and so on have an aggregate value 2, and thus are on the layer at height $1 - \frac{2}{12} = 0.83$; c, g and i are on the layer at height $1 - \frac{6}{12} = 0.5$; and f and the query point q are on the layer at height 0. For simplicity, we denote each layer by the aggregate value x. By the power-law distribution, the probability $p(x)$ that a POI has an aggregate value x is computed by

$$p(x) = \frac{x^{-\beta}}{\zeta(\beta, x_{min})},$$

where

$$\zeta(\beta, x_{min}) = \sum_{i=0}^{\infty}(i + x_{min})^{-\beta}$$

is the Hurwitz zeta function [8]. The expected number of POIs on layer x, which is denoted by $\mathcal{N}(x)$, is computed by

$$\mathcal{N}(x) = \mathcal{N} \cdot p(x),$$

where $\mathcal{N}$ is the total number of POIs. Let the horizontal cross-section of the search region cut by layer x be $\mathcal{D}(q, r_x)$. The radius r_x of $\mathcal{D}(q, r_x)$ is computed by

$$r_x = \frac{h_l - h_x}{h_l} \cdot r_0,$$

where h_x is the height of layer x. Assume that the POIs are uniformly distributed on each layer. We can estimate the expected number of POIs in $\mathcal{D}(q, r_x)$ by $\mathcal{N}(x) \cdot \pi r_x^2$. However, the *boundary effects* cannot be neglected. Boundary effects represent the problem that some parts of the search region lie out of the 3-dimensional unit cube (e.g., when $k = 2$ in the above example). Taking the boundary effects into account, according to [4], the expected number of POIs bounded by $\mathcal{D}(q, r_x)$ is computed by

$$\mathcal{N}(x) \cdot E[S_{\mathcal{D}(q,r_x) \cap U_x}],$$

where $E[S_{\mathcal{D}(q,r_x) \cap U_x}]$ is the expected area that $\mathcal{D}(q, r_x)$ intersects layer x. Assuming that the query point is uniformly distributed, according to [27], we can approximate

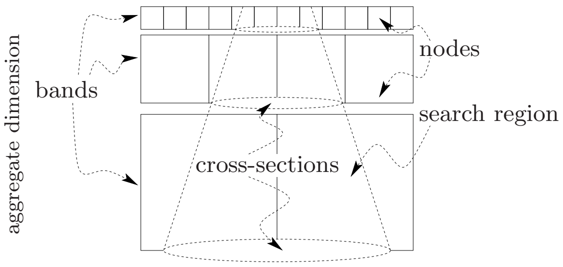


Figure 4: Node accesses estimation example

$E[S_{\mathcal{D}(q,r_x) \cap U_x}]$ by

$$\begin{cases} \left(\sqrt{\pi} \cdot r_x - \frac{\pi r_x^2}{4}\right)^2, & \sqrt{\pi} \cdot r_x < 2 \\ 1, & \textit{otherwise.} \end{cases}$$

Adding up the number of POIs bounded by the cross-section on each layer, $f(p_k)$ can be estimated by solving the following equation:

$$k = \sum_{x=\Omega}^{\infty} \mathcal{N}(x) \cdot E[S_{\mathcal{D}(q,r_x) \cap U_x}],$$

where Ω is the minimum aggregate value.

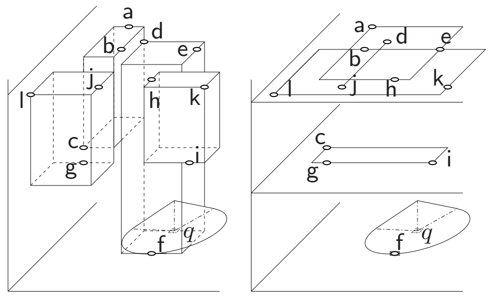
6.3 Estimation of the Number of Node Accesses

We estimate the number of node accesses by computing the number of nodes intersecting the search region. Without loss of generality, we only estimate the number of leaf nodes intersecting the search region since the number of internal nodes is much smaller than the number of leaf nodes. Also, the following analysis applies to internal nodes straightforwardly. The main challenge in the estimation is that the node extents are not uniform along the aggregate dimension due to the power-law distribution. The unit cube is divided into several *bands* along the aggregate dimension (computing the range of each band is detailed below). For example, in Figure 4, each square represents the extents of a node. The squares are small among higher layers and large among lower layers. The nodes of different extents form three bands. We first estimate the node extents and then the number of node accesses in each band.

Following existing cost analyses on the R-tree [12][5][27], we assume that the leaf nodes are of a cubic shape. We estimate the node extents by the extent along the aggregate dimension and the extents along the spatial dimensions. Starting from the top layer x, we proceed downward along the aggregate dimension. When we reach layer y, the node height equals $\Delta h = h_x - h_y$. Meanwhile, according to [5] the node extents along the spatial dimensions equal

$$S_y = \left(1 - \frac{1}{f}\right)\left(\min\left\{\frac{f}{\sum_{i=x}^{y} \mathcal{N}(i)}, 1\right\}\right)^{\frac{1}{2}},$$

where f is the fanout (the average number of entries in a node which typically equals 69% of the node capacity [28]). We obtain the node extents by solving the equation $S_y = \Delta h$ (or $S_y - \Delta h < \epsilon$). We refer to the space from layer x to layer y as a *band* (as shown in Figure 4). We then compute the expected number of nodes accesses in this band. The probability P_y that a node in a band intersects the search region is computed by the Minkowski sum [5] of the node extents S_y and the cross-section $\mathcal{D}(q, r_y)$ cut by the layer y (as illustrated in Figure 4). Taking the boundary effects



(a) (b)
Figure 5: Entry grouping examples

into account, according to [27], P_y can be estimated by

$$P_y = \begin{cases} \left(\dfrac{4L_y - (L_y + S_y)^2}{4(1 - S_y)} \right)^2, & L_y + S_y < 2, \\ 1, & otherwise, \end{cases}$$

where

$$L_y = \left[\sum_{i=0}^{2} \left(\binom{2}{i} \cdot S_y^{2-i} \cdot \frac{\sqrt{\pi}^i}{\Gamma(i/2+1)} \cdot r_y^i \right) \right]^{\frac{1}{2}}.$$

The expected number of node accesses NA_y in this band is thus computed by

$$NA_y = \frac{\sum_{i=x}^{y} \mathcal{N}(i)}{f} \cdot P_y,$$

where $\frac{\sum_{i=x}^{y} \mathcal{N}(i)}{f}$ is the number of nodes in this band. We then proceed with $x = y + 1$ and repeat the above steps until all layers are processed. The expected number of leaf node accesses, denoted by $NA(\alpha, k)$, equals the sum of the number of node accesses computed in each band, i.e.,

$$NA(\alpha, k) = \sum_y NA_y.$$

6.4 Comparison of Entry Grouping Strategies

Based on the above analysis (which is validated by our experiments), we qualitatively compare the three grouping strategies (discussed in Section 5).

If we use the spatial extents to group the entries, the nodes have weak pruning power in the aggregate dimension. The reason is that the nodes will be of a hyper-rectangle shape due to the power-law distribution. For example, in Figure 5(a) the hyper-rectangles represent the nodes. The lower part of such a node may intersect the search region with a high probability. The entries at the top of the unit cube are less likely to contain query results, however, they are accessed if the lower part of the node intersects the search region. The power-law indicates that 80% of the entries are at the top of the unit cube, and hence many nodes will be accessed unnecessarily.

If we use the aggregate distribution to group the entries, the nodes have weak pruning power in the spatial dimensions. This is because the nodes will cover a large space in the spatial dimensions since the spatial proximity is not considered. For example, Figure 5(b) shows the rectangles on each layer representing the nodes. We can see that they have large extents in the spatial dimensions and will be accessed with a high probability provided the height of the search region is greater than the layer containing the node.

The above drawbacks can be avoided when we use the integral 3D strategy. The node extents will follow a power-law-like distribution as shown in Figure 4. The nodes hence retain the pruning power of both spatial and aggregate dimensions. Therefore, the TAR-tree results in much fewer node accesses than alternatives that organizes entries using only the spatial proximity or the aggregate distribution.

7. ENHANCEMENTS FOR THE QUERY

In this section, we propose two enhancement techniques for the kNNTA query. In Section 7.1, we present an algorithm suggesting the least amount of weights to be adjusted that can cause the query results to be changed. In Section 7.2, we present a collective processing scheme to share the index traversal among a batch of queries.

7.1 Suggesting the Minimum Weight Adjustment

New users of the kNNTA query may have difficulty in setting the weights between the spatial distance and aggregate properly. They may adjust the weights to explore different results. It is discouraging if the results remain the same after the weights have been changed. We tackle this problem by suggesting the users the *minimum weight adjustment* that can change the current results (here, changing the results refers to changing the POIs in the kNNTA answer set).

A few existing studies proposed algorithms retaining the top-k results instead of changing the results. For example, Mouratidis et al. [15] proposed an algorithm that computes the *immutable regions* which is defined as the widest range of α_i that preserves the top-k results (assuming that the other weight α_{1-i} is kept constant). Soliman et al. [22] studied finding the maximal hypersphere centered at the weight vector $[\alpha_0, \alpha_1]^T$ such that each vector in the hypersphere preserves (including the order) the top-k results. These algorithms do not apply since they cannot compute the weight adjustment to change the top-k results.

To solve this problem, we first rewrite the ranking function $f(p)$. Let the POIs be ranked in a list. We denote the i^{th} ranked POI by p_i and rewrite the ranking function of p_i by $f(p_i) = \alpha_0 s_{i,0} + \alpha_1 s_{i,1}$, where $s_{i,0} = d(p_i, q)$ and $s_{i,1} = 1 - g(p_i, \mathcal{I}_q)$. For simplicity, we focus on the adjustment of α_0 (since $\alpha_1 = 1 - \alpha_0$). Given a top-k POI p_i ($i \le k$) and a lower ranked POI p_j ($j > k$), where $f(p_i) < f(p_j)$, we obtain a value range of α_0 such that for any α_0' in the range, a ranking function $f'(p)$ defined by α_0' satisfies $f'(p_i) > f'(p_j)$. For example, in the ranking list in Table 3, we have $\alpha_0 = \alpha_1 = 0.5$ and $k = 2$. To let $f'(p_1) > f'(p_3)$, we need $\alpha_0' > \frac{5}{6}$. To let $f'(p_1) > f'(p_6)$, we need $\alpha_0' < \frac{1}{8}$. We refer to the boundary of the range as the *weight adjustment*, denoted by $\gamma_{i,j}$. Let $\delta_{i,j,t} = s_{i,t} - s_{j,t}$ for $t = 0, 1$. When $\delta_{i,j,0} \cdot \delta_{i,j,1} < 0$, $\gamma_{i,j}$ is computed by

$$\gamma_{i,j} = \frac{\delta_{i,j,1}}{\delta_{i,j,1} - \delta_{i,j,0}}.$$

When $\delta_{i,j,0} \cdot \delta_{i,j,1} \ge 0$, we cannot achieve $f'(p_i) > f'(p_j)$ since p_i *dominates* p_j (i.e., $s_{i,t} < s_{j,t}$ for $t = 0, 1$). The minimum weight adjustment (MWA) is the weight adjustments that are nearest to the current weight, i.e., the $\max\{\gamma_{i,j}\}$ or $\min\{\gamma_{i,j}\}$ when $\gamma_{i,j}$ is less or greater than the current weight. For example, in the ranking list in Table 3, to let $f'(p_1)$ be greater than $f'(p_3)$, $f'(p_5)$, $f'(p_6)$, we need $\alpha_0' > \frac{5}{6}$, $\alpha_0' > \frac{20}{29}$, $\alpha_0' < \frac{1}{8}$, and to let $f'(p_2)$ be greater than $f'(p_4)$,

Table 3: Ranking list

POI	$s_{i,0}$	$s_{i,1}$	POI	$s_{i,0}$	$s_{i,1}$
p_1	0.25	0.10	p_4	0.35	0.25
p_2	0.10	0.30	p_5	0.025	0.60
p_3	0.20	0.35	p_6	0.60	0.05

$f'(p_5)$, $f'(p_6)$, we need $\alpha'_0 < \frac{1}{6}$, $\alpha'_0 > \frac{4}{5}$, $\alpha'_0 < \frac{1}{3}$, respectively. The MWA of α_0 is either $\alpha'_0 < \frac{1}{3}$ or $\alpha'_0 > \frac{20}{29}$, since $\frac{1}{3}$ and $\frac{20}{29}$ and are nearest to the current weight 0.5 when the weight adjustment is less and greater than 0.5, respectively. More precisely, the MWA for α_0 comprises two values Γ_l and Γ_u that are computed by:

$$\begin{cases} \Gamma_l = \max\{\gamma_{i,j}\} \text{ for } \delta_{i,j,0} < 0, i \le k, j > k, \\ \Gamma_u = \min\{\gamma_{i,j}\} \text{ for } \delta_{i,j,0} > 0, i \le k, j > k. \end{cases}$$

The MWA will change exactly one of the top-k POIs and keeps the other top-k POIs (the order within top k may change). For example, if we change α_0 to 0.75 in the above example, the new top-2 POIs will be the current p_2 and p_5.

A straightforward way to compute the MWA on the TAR-tree is as follows: After finding the top-k POIs, for each of the top-k POIs p, we continue the BFS until the queue is empty. If the ejected entry e is dominated by p, we continue. Otherwise, we compute and update the (tentative) MWA if e is a leaf entry, or continue the BFS if e is an internal entry. This approach may incur significant cost since it enumerates each of the top-k POIs and has a very weak pruning power on the lower ranked POIs (by checking the dominance).

To overcome this drawback, we propose an approach that makes use of the skyline queries. We observe that: when $\delta_{i,j,0} > 0$ and $\delta_{i,j,0} < 0$, the weight adjustment computed from an entry gives an upper and lower bound on the weight adjustments computed from the child entries, respectively. The reason is, when $\delta_{i,j,0} < 0$ and $\delta_{i,j,0} > 0$,

$$\gamma_{i,j} = \frac{\delta_{i,j,1}}{\delta_{i,j,1} - \delta_{i,j,0}} = \frac{1}{1 - \frac{\delta_{i,j,0}}{\delta_{i,j,1}}} = \frac{1}{1 - \frac{s_{i,0} - s_{j,0}}{s_{i,1} - s_{j,1}}}$$

increases and decreases with the decrease of $s_{j,1}$ or $s_{j,0}$, respectively. Therefore, to compute the MWA, we only need to consider the weight adjustments when interchange the POIs on (i) the skyline of the lower ranked POIs and (ii) the skyline of the top-k POIs with the dominating condition reversed (i.e., p_i dominates p_j if $s_{i,t} > s_{j,t}$ for $t = 0, 1$). Therefore, after finding the top-k POIs, we propose to (i) first compute the skyline of the top-k POIs with the dominating condition reversed, and then (ii) compute the skyline of the lower ranked POIs and (iii) obtain the MWA by the weight adjustments interchanging the POIs on the two skylines. Note that although the proposed TAR-tree is designed for the kNNTA query, it also enables efficient answering of the skyline query, since many skyline algorithms are based on the R-tree (e.g., [18]). It is not difficult to extend the algorithm to compute the weight adjustment that leads to multiple top-k POIs being changed.

7.2 Collective Query Processing

To achieve high scalability when processing multiple kNNTA queries simultaneously, we propose to process kNNTA queries in a batch fashion.

Let c be the number of queries in a batch. We use c priority queues for the BFS of the c queries. In the BFS, we access a node when the front entry is an internal entry. Some front entries in the c queues may be the same (pointing to the same node). For example, if $c = 5$, after we insert a root node containing two entries R_1 and R_2, the front entries of the c queues may be R_1, R_1, R_2, R_2 and R_1. To reduce the number of node accesses, we process the c queues greedily, i.e., the queues containing the most frequent front entry are processed first, which makes the accessed node be shared by the most queries. For instance, in the above example, the node R_1 will be retrieved and the three queues having R_1 as the front entry will be processed first. To further share the aggregate computation on the TIAs in the accessed node, we group the queries together if they have the same query time interval (i.e., the same start time and length) and process the queries as a batch. Such grouping method is effective because in real applications users are usually given only a few options for the query time interval (e.g., one day or one week from now) by default.

Table 4: Data Set

Name	Time	Locations	Check-ins
NYC	05/2008-06/2011	72,626	237,784
LA	02/2009-07/2011	45,591	127,924
GW	02/2009-10/2010	1,280,969	6,442,803
GS	01/2011-07/2011	182,968	1,385,223

8. EXPERIMENTS

In this section, we empirically evaluate the cost analysis, the TAR-tree and two enhancements for the kNNTA query.

Experiments Setup. We use four real-world data sets: NYC, LA, GW and GS. NYC and LA [1] are two LBSNs for the New York City and Los Angeles, respectively (generated from Foursquare tips), GW [7] is the LBSN Gowalla and GS [11] is the LBSN Foursquare (generated from check-ins posted on Twitter). The details of the data sets are listed in Table 4. We implement the R-tree with the R*-tree [3] and the TIA with the Multi-version B-tree. Given the vast memory capacity of modern computers, the R-tree is kept in memory and each TIA is assigned a maximum of 10 buffer slots. To simulate real scenarios, unless otherwise specified, the R-tree node size is set to 1024 bytes (and hence the node capacities are 50 and 36 for 2- and 3-dimensional entries respectively), the epoch length is set to 7 days, and a location needs to have 15, 10, 100 and 50 check-ins for the four data sets respectively to be treated as an effective public POI and indexed. For each data set, we generate 1,000 queries with the query point uniformly sampled from the data set and the query time interval uniformly sampled from $2^0, 2^1, \ldots, 2^9$ days. By default $k = 10$ and $\alpha_0 = 0.3$.

The experiments are conducted on a 64-bit Windows desktop computer with a 3.40GHz Intel(R) Core(TM) i7-2600 CPU and 16GB RAM. All algorithms are implemented in Java. For all sets of experiments (except the validation of the cost analysis), we measure the CPU time and number of node accesses. All presented results are averaged over the 1,000 queries. Due to the space limitation, we only present the results of GW and GS. The results of NYC and LA are consistent with those of GW and GS, and hence are omitted.

8.1 Validation of the Cost Analysis

In this set of experiments, we evaluate the cost analysis by comparing the **estimated** $f(p_k)$ and number of leaf node accesses with the **measured** ones.

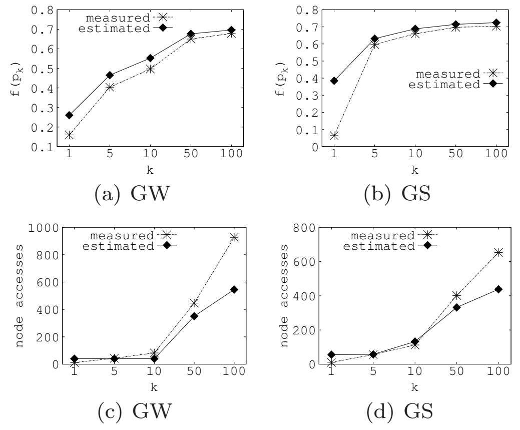


Figure 6: Cost analysis validation by varying k

Varying k. We first evaluate the analysis by varying k from 1 to 100. The results are plotted in Figure 6. Figures 6(a) and 6(b) show the comparison of the estimated and measured $f(p_k)$. We can see that $f(p_k)$ increases with the increase of k as expected. The estimates are very close to the actual values when $k \geq 5$. The estimate is less accurate when $k < 5$, especially on GS, which is due to the large variance of $f(p_k)$ when $k < 5$ and only a few POIs have large aggregate values. Figures 6(c) and 6(d) present the comparison of the estimated and measured number of node accesses. We can see that with the increase of k the number of leaf node accesses increases and the estimates exhibit the same growing trend as the actual values. When $k \leq 50$, the estimates approximate the measured values very well. When $k > 50$, the estimation is slightly inaccurate due to that the number of POIs computed by the power-law is less accurate when x is close to $\hat{x}_{min}$ and it happens more frequently when k is larger. This problem can be addressed by collecting more data or introducing a more complex piece-wise fitting.

Varying α_0. Next, we evaluate the analysis by varying α_0 from 0.1 to 0.9. The results are plotted in Figure 7. Figures 7(a) and 7(b) depict the comparison of the estimated and measured $f(p_k)$. For all values of α_0, the estimates are almost identical to the actual values. Figures 7(c) and 7(d) illustrate the comparison of the estimated and measured number of node accesses. We can see that the number of node accesses increases moderately with the increase of α_0. The estimates fluctuate closely around the actual values. When α_0 is close to 0.9, the estimates show an opposite growing trend to the actual values. This is due to the same problem caused by the error of the power-law fitting when x is close to $\hat{x}_{min}$, and can also be addressed by the same approaches. Overall, the cost analysis is accurate and can strongly indicate the query processing cost.

8.2 Performance of the TAR-tree

In this set of experiments, we evaluate the performance of the TAR-tree. We compare the **TAR-tree** with the two alternatives (discussed in Section 5) using the spatial extents and the aggregate distribution to group the entries, respectively. We refer to the two alternatives as the **IND-spa** and **IND-agg**, respectively. We also compare the TAR-tree with the straightforward approach (scanning the aggregate values and POIs) to measure the query processing speed up, which is referred to as the **baseline**.

Effect of the LBSN Growing with Time. First, we evaluate the performance by simulating the growth of the LBSN with time. We take a snapshot on each data set at $20\%, 40\%, \ldots, 100\%$ of the time. The results are plotted in Figure 8. Figures 8(a) and 8(b) show the CPU time of different approaches. We can see that the TAR-tree runs several times faster than the IND-spa and IND-agg. The TAR-tree also runs greatly faster than the baseline. With the growth of the LBSN, the performance of the TAR-tree may slightly fluctuate (as shown in Figure 8(b)). This is due to that the TAR-tree does not adjust promptly to adapt to the current LBSN. To address the problem, we can reinsert part of the entries periodically or rebuild the TAR-tree when the performance degrades below some threshold.

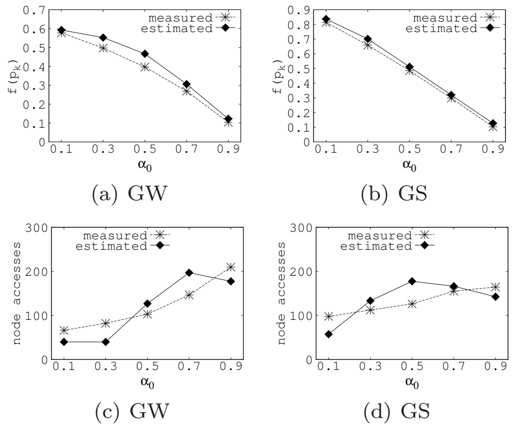


Figure 7: Cost analysis validation by varying α_0

Figures 8(c) and 8(d) present the number of node accesses against the growth of the LBSN. The TAR-tree consistently has the smallest number of node accesses and outperforms the IND-spa and IND-agg by a significant margin on both data sets. On GW, the TAR-sap incurs the largest number of node accesses, while on GS the IND-agg is the worst. This indicates that unlike the TAR-tree, the performance of IND-spa and IND-agg is unstable across different data sets. On GW, the number of node accesses in the TAR-tree marginally decreases with the growth of the LBSN. This implies that the TAR-tree performs better when there are sufficient aggregate information.

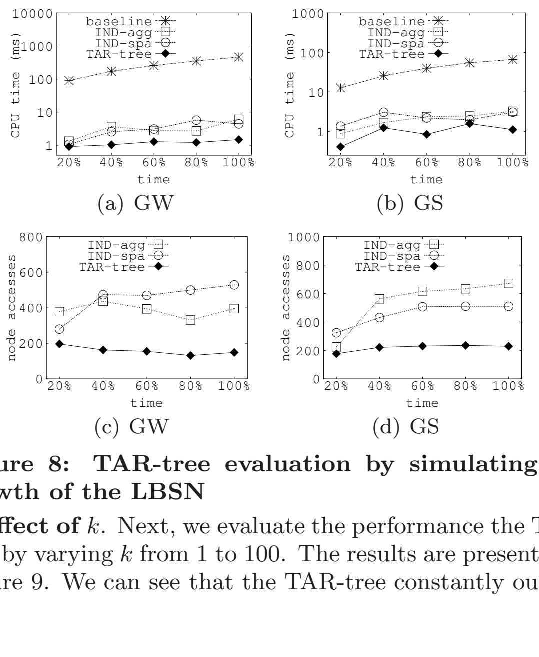
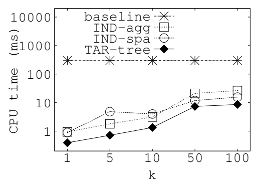
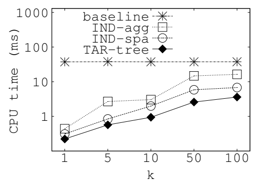


Figure 8: TAR-tree evaluation by simulating the growth of the LBSN

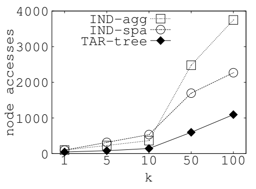
Effect of k. Next, we evaluate the performance the TAR-tree by varying k from 1 to 100. The results are presented in Figure 9. We can see that the TAR-tree constantly outper-



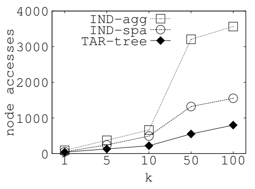
(a) GW



(b) GS

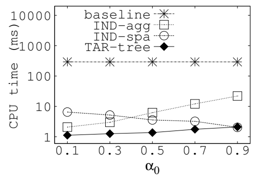


(c) GW

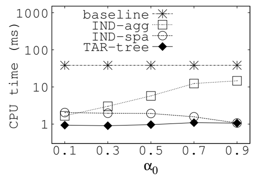


(d) GS

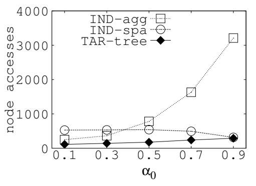
Figure 9: TAR-tree evaluation by varying k



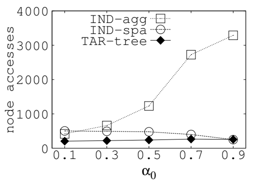
(a) GW



(b) GS

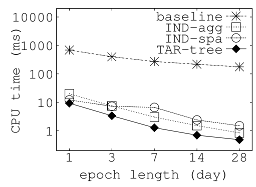


(c) GW

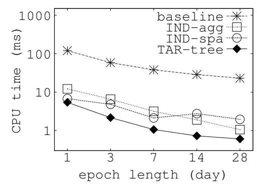


(d) GS

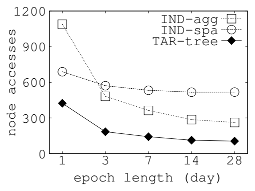
Figure 10: TAR-tree evaluation by varying α_0



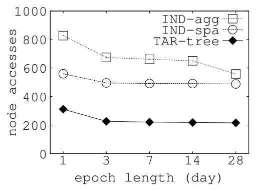
(a) GW



(b) GS

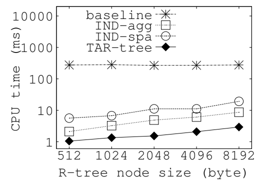


(c) GW

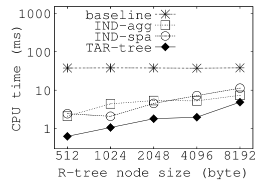


(d) GS

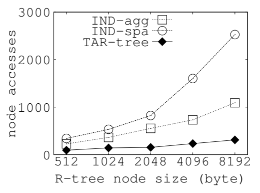
Figure 11: TAR-tree evaluation by varying the epoch length



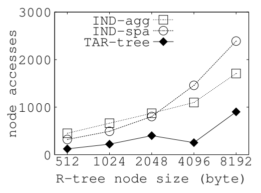
(a) GW



(b) GS



(c) GW



(d) GS

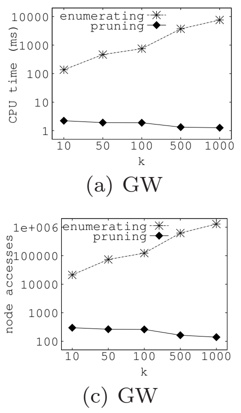
Figure 12: TAR-tree evaluation by varying the R-tree node size

forms all the other approaches. With the increase of k, the CPU time and number of node accesses of all approaches increase. This, as indicated by the cost analysis, is because the search region expands with the increase of k and hence accesses more nodes. When $k \geq 50$, as Figure 9(b) shows, the performance of the IND-agg deteriorates rapidly and its CPU time is comparable to that of the baseline. Figures 9(c) and 9(d) show the number of node accesses. We can see that when $k > 10$, the IND-spa and IND-agg have a much larger growth rate than that of the TAR-tree. This also confirms the cost analysis (in Section 6.4) that the expanding of the search region has a larger impact on IND-spa and IND-agg.

Effect of α_0. Figure 10 plots the performance of different approaches when the value of α_0 is varied from 0.1 to 0.9. When α_0 approaches 1, the importance of the spatial distance increases in the kNNTA query. Figures 10(a) and 10(b) show the CPU time and we can see that when α_0 approaches 1, the performance of the IND-spa and IND-agg decreases and increases, respectively. This is because the IND-spa and IND-agg are optimized for the spatial and aggregate dimensions, respectively. The performance of the TAR-tree is almost unaffected by the changing of α_0 and the TAR-tree keeps running the fastest. Even when $\alpha_0 = 0.1$ and 0.9, for which the IND-agg and IND-spa are supposed to have a good advantage, the TAR-tree still performs no worse than the IND-agg and IND-spa, respectively. Figures 10(a) and 10(b) present similar results on the number of node accesses. We can see that when α_0 approaches 1, the number of node accesses in the IND-agg increases radically. This is because the height of the search region grows rapidly with the increase of α_0, and for the IND-agg, a node is accessed (with a high probability) as long as the layer containing the node is less than or equal to the height of the search region.

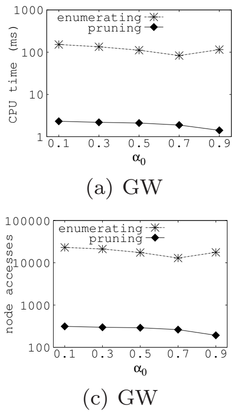
Effect of the Epoch Length. We now proceed to evaluate the performance by varying the parameters of the TAR-tree. First, we vary the epoch length from 1 to 28 days and present the results in Figure 11. Figures 11(a) and 11(b) show that the CPU time of all approaches (including the baseline) decreases with the increase of the epoch length. This is because fewer values are added up to obtain the aggregate. Figures 11(c) and 11(d) show the number of node accesses and we can see that the longer the epoch length is, the fewer node accesses the TAR-tree needs to process the query. The reason is that a longer epoch length strengthens the pruning power of the TAR-tree because the aggregate of a parent node is closer to the maximum aggregate computed from its child nodes. For all epoch lengths, the TAR-tree outperforms the other approaches both in the CPU time and number of node accesses.

Effect of the R-tree Node Size. Next, we vary the R-tree node size from 512 to 8192 bytes and present the results in Figure 12. As shown in Figures 12(a) and 12(b), the CPU time of the TAR-tree increases almost linearly with the increase of the node size. This is because the number of entries in a node grows linearly as the node size grows and similar operations are performed on each entry. Fig-


(a) GW (b) GS

(c) GW (d) GS

Figure 13: Computing the MWA by varying k


(a) GW (b) GS

(c) GW (d) GS

Figure 14: Computing the MWA by varying α_0

ures 12(c) and 12(d) show that the number of node accesses of all approaches increases with the growth of the node size. The IND-spa has the largest growth rate and the TAR-tree has the smallest growth rate. The reason is that with the increase of the node size, the node represent a larger spatial region, and thus has a relatively weak pruning power in spatial extents. Under all settings, the TAR-tree consistently outperforms all the other approaches.

8.3 Performance of the Weight Adjustment Algorithm

In this set of experiments, we compare the performance of the proposed weight adjustment algorithm with the straightforward approach. We refer to the proposed algorithm and the straightforward approach as **pruning** and **enumerating**, respectively.

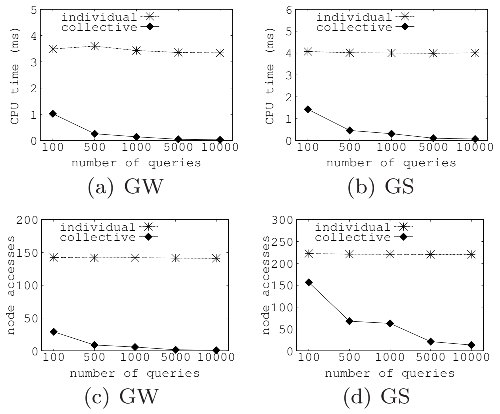
Varying k. We first evaluate the algorithm by varying k from 10 to 1000. We plot the results in Figure 13. From Figures 13(a) and 13(b), we can see that pruning runs orders of magnitude faster than enumerating. The performance of enumerating degrades rapidly as k grows. This is because each top-k result is enumerated and computed against the lower ranked POIs. The CPU time of the pruning algorithm decreases marginally with the increase of k. This is because computing the skyline of the lower ranked POIs takes much less time and pays off the time spent on computing the skyline of the top-k POIs. Figures 13(c) and 13(d) show consistent results on the number of node accesses except that the number of node accesses decreases marginally faster than the CPU time since it incurs no node accesses to compute the skyline of the top-k POIs.

Varying α_0. Next, we evaluate the algorithm by varying α_0 from 0.1 to 0.9. The results are presented in Figure 14. As Figures 14(a) and 14(b) show, the CPU time of enumerating first decreases and then increases as α_0 grows. Since checking the dominance is the only pruning technique used by this approach, it indicates that the pruning power of the technique is weakest when α_0 is around 0.1 or 0.9. The CPU time of the pruning algorithm has an opposite growing trend to enumerating. This indicates that it is more efficient to compute the skyline when the weight is skewed. Figures 14(c) and 14(d) present consistent results on the number of node accesses. In all settings, the proposed algorithm outperforms the baseline by a significant margin.

8.4 Performance of the Collective Processing Scheme

In the last set of experiments, we evaluate the collective processing scheme (**collective**) against the approach to processing the query individually (**individual**). To investigate the effect of memory buffering on processing the query individually, we assign no buffer to the TIAs.

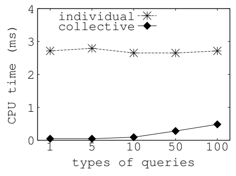
Varying the Number of Queries. Figure 15 presents the CPU time and the node accesses as a function of the number of queries. From Figures 15(a), 15(b) and Figures 15(c), 15(d), we can see that for the collective processing scheme, the more queries are processed collectively, the shorter the average processing time is and the smaller number of node accesses we need, respectively. This is because more queries share the index traversal. We can also see that when processing the query individually, changing the number of queries has little effect on either the CPU time or the number of node accesses. The collective processing scheme constantly outperforms processing the query individually by a big margin.


(a) GW (b) GS

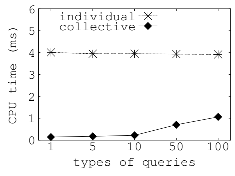
(c) GW (d) GS

Figure 15: Collective processing by varying the number of queries

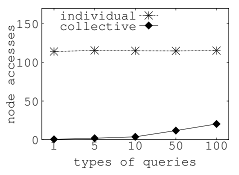
Varying the Number of Query Types. Next, we evaluate the collective processing scheme by varying the number of query time intervals (i.e., query types) from 1 to 100. Figure 16 presents the results. Since the queries are grouped by the query time interval, the efficiency of the collective processing scheme will decline when the number of query time interval increases. As Figures 16(a) and 16(b) show, the efficiency of the collective processing scheme does not degrade too much when there are more than 10 types of queries. The collective processing scheme keeps running several times faster than processing the queries individually. Figures 16(c) and 16(d) present consistent results on the number of node accesses. In all settings, the collective processing scheme outperforms the baseline by a significant margin.



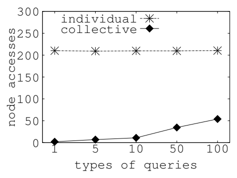
(a) GW



(b) GS



(c) GW



(d) GS

Figure 16: Collective processing by varying the number of query types

9. CONCLUSIONS

We proposed a new type of queries called the k-nearest neighbor temporal aggregate query, which provides highly customized POI retrieval by integrating the spatial distance and a temporal aggregate on a certain attribute. We designed a novel index called the TAR-tree by integrating both types of information to effectively group the entries, and therefore can support efficient processing of the kNNTA query. We performed a detailed analysis on the cost of query processing using the TAR-tree. The analysis shows that the TAR-tree has a stronger pruning power than alternatives. The accuracy of the cost analysis is validated by empirical experiments. Furthermore, we proposed two enhancements for the kNNTA query: (i) To assist users explore different results, we devised an efficient algorithm suggesting the minimum weight adjustment that can change the query results. (ii) To handle large number of queries, we proposed an effective collective processing scheme to share the index traversal among a batch of queries. We conducted extensive experiments on real-world data sets. The results validate the efficiency of the TAR-tree and the effectiveness of the two enhancements for the query.

Acknowledgments. This work is supported by Australian Research Council (ARC) Discovery Project DP130104587 and Australian Research Council (ARC) Future Fellowships Project FT120100832.

10. REFERENCES

[1] J. Bao, Y. Zheng, and M. F. Mokbel. Location-based and preference-aware recommendation using sparse geo-social networking data. In *SIGSPATIAL*, pages 199–208, 2012.

[2] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion b-tree. *VLDBJ*, 5(4):264–275, 1996.

[3] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.

[4] S. Berchtold, C. Böhm, D. A. Keim, and H.-P. Kriegel. A cost model for nearest neighbor search in high-dimensional data space. In *PODS*, pages 78–86, 1997.

[5] C. Böhm. A cost model for query processing in high dimensional data spaces. *TODS*, 25(2):129–178, 2000.

[6] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.

[7] E. Cho, S. A. Myers, and J. Leskovec. Friendship and mobility: User movement in location-based social networks. In *KDD*, pages 1082–1090, 2011.

[8] A. Clauset, C. R. Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. *SIAM Review*, 51(4):661–703, 2009.

[9] G. Cong, C. S. Jensen, and D. Wu. Efficient retrieval of the top-k most relevant spatial web objects. *PVLDB*, 2(1):337–348, 2009.

[10] H. G. Elmongui, M. F. Mokbel, and W. G. Aref. Continuous aggregate nearest neighbor queries. *GeoInformatica*, 17(1):63–95, 2013.

[11] H. Gao, J. Tang, and H. Liu. gscorr: Modeling geo-social correlations for new check-ins on location-based social networks. In *CIKM*, pages 1582–1586, 2012.

[12] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *TODS*, 24(2):265–318, 1999.

[13] J. Huang, Z. Wen, J. Qi, R. Zhang, J. Chen, and Z. He. Top-k most influential locations selection. In *CIKM*, 2011.

[14] H. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang. idistance: An adaptive b+-tree based indexing method for nearest neighbor search. *TODS*, 30(2):364–397, 2005.

[15] K. Mouratidis and H. Pang. Computing immutable regions for subspace top-k queries. *PVLDB*, 6(2):73–84, Dec. 2012.

[16] S. Nutanong, R. Zhang, E. Tanin, and L. Kulik. The v*-diagram: a query-dependent approach to moving knn queries. *PVLDB*, 1(1):1095–1106, 2008.

[17] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient olap operations in spatial data warehouses. In *SSTD*, 2001.

[18] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD*, pages 467–478, 2003.

[19] D. Papadias, Y. Tao, K. Mouratidis, and C. K. Hui. Aggregate nearest neighbor queries in spatial databases. *TODS*, 30(2):529–576, 2005.

[20] J. Qi, R. Zhang, L. Kulik, D. Lin, and Y. Xue. The min-dist location selection query. In *ICDE*, pages 366–377, 2012.

[21] J. Qi, R. Zhang, Y. Wang, A. Y. Xue, G. Yu, and L. Kulik. The min-dist location selection and facility replacement queries. *World Wide Web*, 17(6):1261–1293, 2014.

[22] M. A. Soliman, I. F. Ilyas, D. Martinenghi, and M. Tagliasacchi. Ranking with uncertain scoring functions: Semantics and sensitivity measures. In *SIGMOD*, 2011.

[23] J. Sun, D. Papadias, Y. Tao, and B. Liu. Querying about the past, the present, and the future in spatio-temporal databases. In *ICDE*, pages 202–213, 2004.

[24] Y. Tao, G. Kollios, J. Considine, F. Li, and D. Papadias. Spatio-temporal aggregation using sketches. In *ICDE*, pages 214–225, 2004.

[25] Y. Tao and D. Papadias. Range aggregate processing in spatial databases. *TKDE*, 16(12):1555–1570, 2004.

[26] Y. Tao and D. Papadias. Historical spatio-temporal aggregation. *TOIS*, 23(1):61–102, 2005.

[27] Y. Tao, J. Zhang, D. Papadias, and N. Mamoulis. An efficient cost model for optimization of nearest neighbor search in low and medium dimensional spaces. *TKDE*, 16(10):1169–1184, 2004.

[28] Y. Theodoridis and T. Sellis. A model for the prediction of r-tree performance. In *PODS*, pages 161–171, 1996.

[29] R. Zhang, N. Koudas, B. C. Ooi, and D. Srivastava. Multiple aggregations over data streams. In *SIGMOD*, pages 299–310, 2005.

[30] R. Zhang, J. Qi, M. Stradling, and J. Huang. Towards a painless index for spatial objects. *TODS*, 39(3):19, 2014.

[31] Y. Zheng. Location-based social networks: Users. In *Computing with Spatial Trajectories*, pages 243–276. 2011.

[32] Y. Zheng, L. Capra, O. Wolfson, and H. Yang. Urban computing: concepts, methodologies, and applications. *Transaction on Intelligent Systems and Technology*, 2014.

Interactive Path Query Specification on Graph Databases

Angela Bonifati Radu Ciucanu Aurélien Lemay
University of Lille & INRIA, France
{angela.bonifati, radu.ciucanu, aurelien.lemay}@inria.fr

ABSTRACT

Graph databases are becoming pervasive in several application scenarios such as the Semantic Web, social and biological networks, and geographical databases, to name a few. However, specifying a graph query is a cumbersome task for non-expert users because graph databases (i) are usually of large size hence difficult to visualize and (ii) do not carry proper metadata as there is no clear distinction between the instances and the schemas. We present GPS, a system for interactive path query specification on graph databases, which assists the user to specify path queries defined by regular expressions. The user is interactively asked to visualize small fragments of the graph and to label nodes of interest as positive or negative, depending on whether or not she would like the nodes as part of the query result. After each interaction, the system prunes the uninformative nodes i.e., those that do not add any information about the user's goal query. Thus, the system also guides the user to specify her goal query with a minimal number of interactions.

1. INTRODUCTION

Graph databases [8] are becoming pervasive in several application scenarios such as the Semantic Web, social and biological networks, and geographical databases, to name a few. Many mechanisms have been proposed to query a graph database, which, although being very expressive, are difficult to understand by non-expert users who are unable to specify their queries with a formal syntax.

The problem of *assisting non-expert users to specify their queries* has been recently raised by Jagadish et al. [6, 7]. More concretely, they have observed that "constructing a database query is often challenging for the user, commonly takes longer than the execution of the query itself, and does not use any insights from the database". While they have mentioned these problems in the context of relational databases, we argue that they become even more difficult to tackle for graph databases. Indeed, graph databases usually do not carry proper metadata as there is no clear distinction between the instances and the schemas. The absence of metadata and the difficulty of visualizing possibly large graphs make unfeasible traditional query specification paradigms for non-expert users e.g., query by example [9].

In this paper, we address the problem of assisting non-expert users to specify their graph queries and propose GPS, "a system for interactive Graph Path query Specification". The user is interactively asked to visualize small fragments of the graph and to label nodes of interest as *positive* or *negative* examples, depending on whether or not she would like the nodes as part of the query result. After each interaction, the system prunes the uninformative nodes i.e., those nodes that do not provide any useful information about the user's goal query. Thus, the system also guides the user to specify her goal query with a minimal number of interactions.

In [2], we have studied the theoretical challenges of such a scenario and empirically shown the improvements of using an interactive approach on biological and synthetic datasets. As a natural extension, we are interested next in applying our algorithms to scenarios where human users provide the positive and negative examples. Both in [2] and in this demo, we focus on the class of path queries defined by regular expressions, where a node is selected if it has a path in the language of a given regular expression. The objective of this demo is thus to let real users interact with GPS to specify different path queries that they could have in mind, while minimizing the number of interactions with the system.

The rest of the paper is organized as follows. In Section 2 we present some key ingredients of our system via a motivating example, while in Section 3 we describe our demonstration scenario. Due to space restrictions, in this paper we provide only a glimpse of the techniques employed by GPS. However, we refer to our full research paper [2] for algorithmic details and for more elements of related work.

2. SYSTEM OVERVIEW

In this section we present a brief overview of our system. To this purpose, we first introduce a *motivating example.* Then, we describe the *interactive scenario* for path query specification on graph databases.

Motivating example

We depict in Figure 1 a graph representing a geographical database having as nodes the neighborhoods of a city area (N_1 to N_6), along with cinemas (C_1 and C_2), and restaurants (R_1 and R_2) in such neighborhoods. The edges represent public transportation facilities from a neighborhood to another (using labels *tram* and *bus*), along with other

10.5441/002/edbt.2015.44

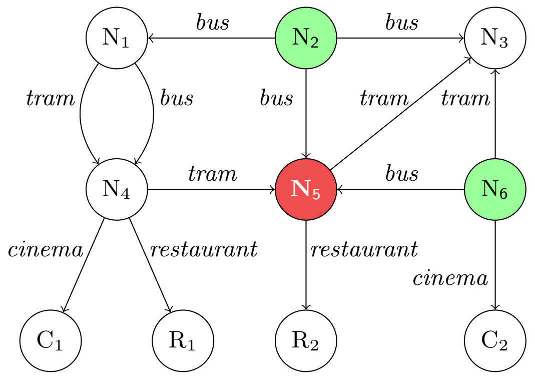
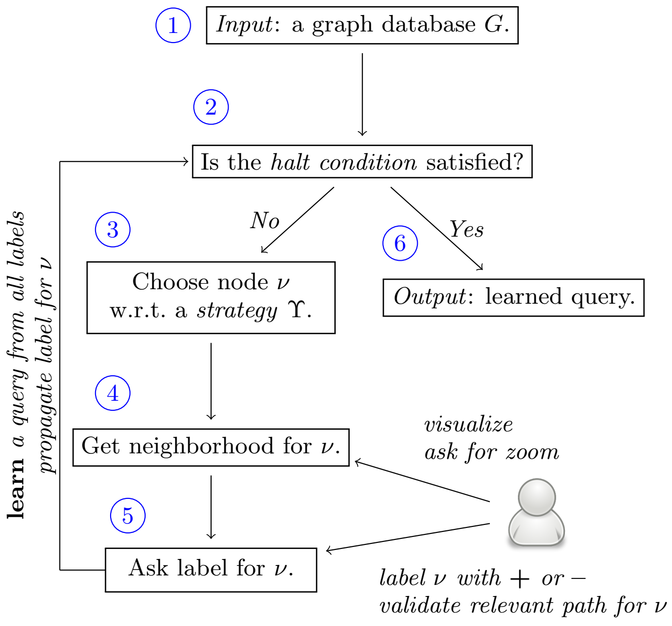


Figure 1: A geographical graph database.



1
Input: a graph database G.
2
Is the halt condition satisfied?
No
Yes
3
Choose node ν w.r.t. a strategy ϒ.
6
Output: learned query.
4
Get neighborhood for ν.
5
Ask label for ν.
visualize
ask for zoom
label ν with + or −
validate relevant path for ν
learn a query from all labels
propagate label for ν

Figure 2: Interactive scenario.

kind of facilities (using labels *cinema* and *restaurant*). For instance, the graph indicates that one can travel by bus between the neighborhoods N_2 and N_3, that in the neighborhood N_4 there is a cinema C_1, and so on. Next, imagine that a user wants to know from which neighborhoods in the city she can reach cinemas via public transportation. These neighborhoods can be retrieved using a *path query* defined by the following *regular expression*:

$$q = (tram + bus)^* \cdot cinema$$

The query q selects the nodes N_1, N_2, N_4, and N_6 as they are entailed by the following *paths* in the graph:

$$N_1 \xrightarrow{tram} N_4 \xrightarrow{cinema} C_1,$$
$$N_2 \xrightarrow{bus} N_1 \xrightarrow{tram} N_4 \xrightarrow{cinema} C_1,$$
$$N_4 \xrightarrow{cinema} C_1,$$
$$N_6 \xrightarrow{cinema} C_2.$$

We assume that the user is not familiar with any formal syntax of query languages, while she still wants to specify the above query on the graph database in Figure 1 by providing examples of the query result. In particular, she would positively or negatively label some graph nodes according to whether or not they would be selected by the targeted query. For instance, the user could label the nodes N_2 and N_6 as *positive examples*, and the node N_5 as a *negative example*, thus willing to have all nodes but the last as part of the query result. Indeed, there is no path starting in N_5 through which the user can reach a cinema, while there are paths for the first two nodes. We also observe that the query q above is *consistent* with the user's examples because q selects all positive examples and none of the negative ones.

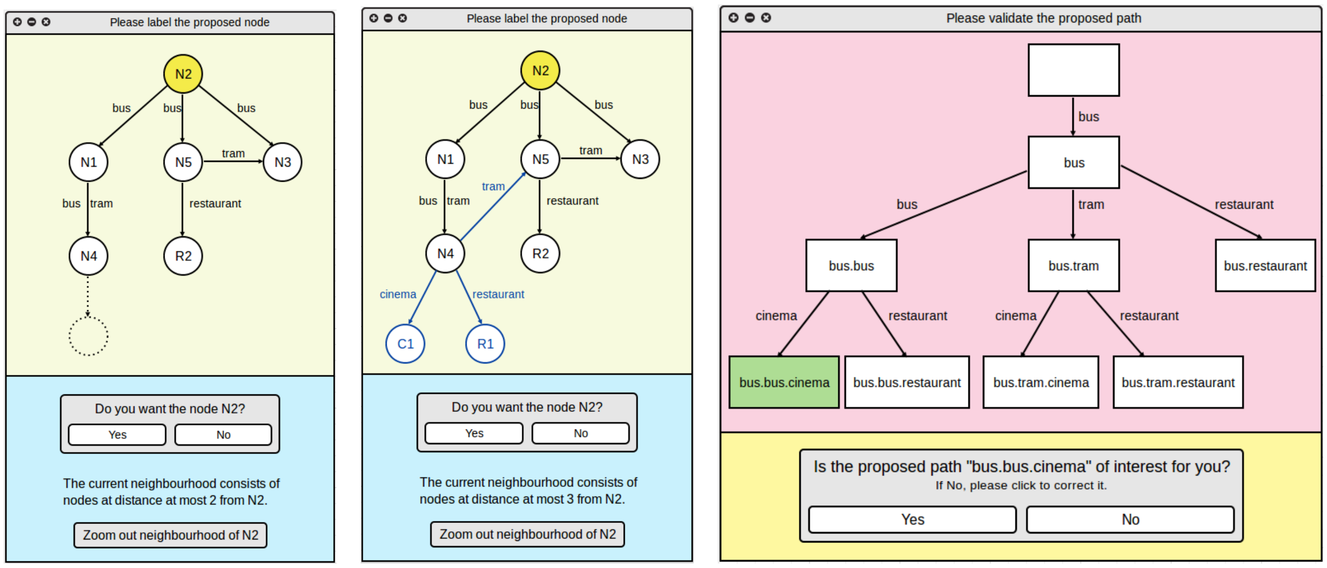
To construct a query that is consistent with the examples provided by the user, we have proposed in [2] a *learning algorithm* that essentially consists of the following two steps: (i) for each positive example, find a path that is not covered by any negative, and (ii) construct an automaton recognizing precisely the paths found at the previous step and generalize it by state merges while no negative example is covered. By continuing on our running example, take the graph database from Figure 1, the positive examples N_2 and N_6, and the negative example N_5. Assuming that at step (i) we have found the paths *bus·tram·cinema* and *cinema* for N_2 and N_6, respectively, by generalizing the disjunction of these two paths we are able to construct the aforementioned query q, which corresponds to the user's goal query. In the next section, while describing the interactive scenario, we explain in more details how our system is able to find path queries via simple user interactions.

Interactive scenario

Even though our demonstration scenario consists of several types of interactions with the user (cf. Section 3), in this section we concentrate exclusively on the core of GPS, the *interactive scenario* for path query specification. This scenario is inspired by the well-known framework of *learning with membership queries* [1]. Recently, we have formalized it as a general paradigm for learning queries on big data [3] and also employed it for learning join queries on relational databases [4, 5]. In Figure 2, we depict the current instantiation for path queries on graphs and we detail next its different steps.

(1) (2) We consider as input a graph database G. Initially, we assume an empty set of examples that we enrich via simple interactions with the user. The interactions continue until a *halt condition* is satisfied. A natural condition is to stop the interactions when there is exactly one consistent query with the current set of examples. However, we also allow weaker conditions e.g., the user may stop the process earlier if she is satisfied by some candidate query proposed at some intermediary stage during the interactions.

(3) We propose nodes of the graph to the user according to a *strategy* Υ i.e., a function that takes as input a graph G and a set of examples S, and returns a node from G. Since our goal is to minimize the amount of effort needed to learn the user's goal query, a smart strategy should avoid proposing to the user those nodes that do not bring any information to the learning process. Intuitively, a node is uninformative if all its paths are covered by negative nodes. A good practical strategy should have two essential properties: (i) be time-efficient i.e., the user does not have to wait too much


Please label the proposed node

Do you want the node N2?

The current neighbourhood consists of nodes at distance at most 2 from N2.

Zoom out neighbourhood of N2

Please label the proposed node

Do you want the node N2?

The current neighbourhood consists of nodes at distance at most 3 from N2.

Zoom out neighbourhood of N2

Please validate the proposed path

Is the proposed path "bus.bus.cinema" of interest for you?

If No, please click to correct it.

Figure 3: Interactions with the user for node labeling and path validation.
(a) Proposing node N_2 to the user and showing the neighborhood of nodes at distance at most 2.
(b) Proposing node N_2 to the user and showing the neighborhood of nodes at distance at most 3.
(c) Proposing a path of node N_2 for user validation and showing all paths of N_2 of length at most 3.

between two consecutive interactions, and (ii) attempt to minimize the number of user interactions by asking the user to only label the most informative nodes. In [2], we have developed such strategies, which intuitively seek the nodes having an important number of paths that are shorter than a fixed bound and not covered by any negative node.

④ ⑤ ⑥ A node by itself does not carry enough information to allow the user understand whether it is part of the query result or not. Therefore, we have to enhance the information of a node by zooming out on its *neighborhood* before actually showing it to the user. This step has the goal of producing a small, easy to visualize fragment of the initial graph, which possibly contains the nodes that the user would label as positive or negative.

In our system, we initially compute the neighborhood of a node ν as the graph consisting of all nodes and edges at distance at most 2 from ν. For instance, given the graph database from Figure 1, let us assume that we want to ask the user to label the node N_2. Thus, the user is first presented with the graph in Figure 3(a). Notice that we have depicted by "…" the parts of the graph that are reachable from the current node N_2, but are not shown because they are not in the current neighborhood. The user can label the proposed node as a positive or negative example (i.e., answer "Yes/No") or she may ask for zooming out the neighborhood to be able to decide whether or not the node is of interest for her. For instance, if the user decides to zoom out the neighborhood of N_2, we show her the graph from Figure 3(b), where we highlight (by drawing in blue) the nodes and edges that have been added w.r.t. the previously presented graph fragment. On our example, the user is able now to see that she can reach a cinema from N_2 and thus to label this proposed node as a positive example.

Next, if the user has labeled a given node as a positive example, we want to find out which of the paths of that node is of interest for her. For this purpose, the system builds all paths of the current node that are not yet covered by negative examples and of length at most the size of the last neighborhood. We present these paths to the user as a prefix tree and we highlight the path that the system believes is the path of interest for her. The user can thus validate this path or correct it further by choosing a different path. In Figure 3(b) of our running example, we have shown the neighborhood of size 3 of the node N_2, which is the node that the user has labeled as a positive example. Consequently, the system shows to the user the paths of N_2 of length at most 3 in the prefix tree of Figure 3(c). The system highlights the path *bus·bus·cinema* as a candidate path of interest for the user because (i) it has length equal to 3 and (ii) the system inferred that a path of this length better fits the user's will as the latter zoomed out the neighborhood of length 2 in Figure 3(a).

After path validation by the user, the system seamlessly propagates to the rest of the graph the labels provided by the user at this stage, while at the same time pruning the nodes that become uninformative. Our learning algorithm outputs in polynomial time either a query q consistent with all labels provided by the user, or instead the next node to label if such a query cannot be constructed efficiently. We have shown in [2] that constructing in polynomial time a query consistent with the examples is not always possible, but, after a certain number of examples (this number being polynomial in the size of the query), the learning algorithm is guaranteed to return in polynomial time a query equivalent to the user's goal query.

When the halt condition is satisfied, we return the latest output of the learning algorithm to the user. In particular, the halt condition may take into account an intermediary learned query q e.g., when the user is satisfied by the output of q on the instance and wants to stop the interactions.

3. DEMONSTRATION SCENARIO

The demonstration scenario consists of three parts. First, we would like that the attendee appreciates the difficulties that one can encounter when labeling directly the graph database instance. Next, we propose to the attendee an interactive scenario where she is prompted with small fragments of the graph that can be easily visualized. On these fragments, the user can label nodes as positive or negative examples and the system infers her queries. Third, we present the core of our system, where we additionally show to the user the set of relevant paths entailed by the positive examples for further validation. This extra step guarantees that the system generalizes the interesting paths for the user and thus the constructed query indeed corresponds to what the user had in mind. In the demo, we plan to show our algorithms on real geographical data. Such data combines the information about networks of public transportation in France (e.g., Transpole[1]) with other facilities in the spirit of our motivating example.

Static labeling

To illustrate why it is important to use an interactive approach in proposing nodes of interest to the user, we progressively show to the attendees the different types of interactions our system can handle. In this first part of the demonstration, we let the attendee visualize the graph and label nodes of interest in the order she prefers. Then, the system proposes a query consistent with the provided examples or, alternatively, points out that the labeled nodes are inconsistent. The attendee must observe that this kind of approach is not user-friendly as the user is asked to (i) possibly visualize a large graph database instance and (ii) inspects interesting fragments by herself. This clearly requires more effort than visualizing small fragments of the graph and simply answering "Yes/No" to nodes proposed by the system. However, we think that it is still important to show this static labeling scenario to the user to appreciate the differences with respect to the interactive scenario, which we discuss next. Moreover, the static labeling scenario is the only one where we let the user to make mistakes by labeling nodes inconsistently because in the other scenarios we show informative nodes only hence any labeling is consistent.

Interactive labeling (without path validation)

In this part of the demonstration, we present the interactive scenario illustrated in the previous section, but without including the step of path validation. For instance, at each interaction the system computes the most informative node and shows it to the user together with its neighborhood as in Figure 3(a). Then, the user may label it as a positive or negative example, or she can ask for zooming out the neighborhood, which yields a graph as in Figure 3(b). When the user labels a proposed node as a positive example, the system computes for that node a path that is not covered by any negative example, the latter path being used by the learning algorithm afterwards. The goal of this scenario is to show the importance of the step of path validation. Although the interactive scenario without this step finally produces a query that is consistent with the examples provided by the user, this query is not necessarily always the query that the user expects. As an example, on the graph and the labeled examples in Figure 1, notice that the query *bus* selects both positive examples C_2 and C_6, and not the negative example C_5. This query is clearly not the user's goal query. Therefore, even though the user has to perform an additional click to validate or to correct the path of interest for a positive node, this step is necessary to make sure that the learned query is constructed using for each positive node the paths of interest for her.

Interactive labeling (with path validation)

In this last part of the demonstration, we illustrate the core of our system i.e., the interactive scenario described in Section 2. As a difference w.r.t. the second demonstration scenario, the user can now additionally choose as in Figure 3(c) the path of interest instead of letting the learning algorithm choose such a path. The goal of this third scenario is to show the actual difference between "learning" a query that is consistent with the node examples provided by the user and assisting the user to "specifying" her query, also via node examples. When the user also validates the paths of interest for each positive node, this guarantees that the system generalizes the interesting paths for the user and the constructed query is indeed the user's goal query. In conclusion, by using GPS, a non-expert user desiring to query a graph database has neither to visualize all the graph that can be potentially large, nor to look by herself for interesting nodes, as the system guides her throughout small, easy to visualize fragments of the graph and prompt her with nodes to label on these fragments.

4. REFERENCES

[1] D. Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1988.

[2] A. Bonifati, R. Ciucanu, and A. Lemay. Learning path queries on graph databases. In *EDBT*, 2015.

[3] A. Bonifati, R. Ciucanu, A. Lemay, and S. Staworko. A paradigm for learning queries on big data. In *Data4U*, pages 7–12, 2014.

[4] A. Bonifati, R. Ciucanu, and S. Staworko. Interactive inference of join queries. In *EDBT*, pages 451–462, 2014.

[5] A. Bonifati, R. Ciucanu, and S. Staworko. Interactive join query inference with JIM. *PVLDB*, 7(13):1541–1544, 2014.

[6] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu. Making database systems usable. In *SIGMOD Conference*, pages 13–24, 2007.

[7] A. Nandi and H. V. Jagadish. Guided interaction: Rethinking the query-result paradigm. *PVLDB*, 4(12):1466–1469, 2011.

[8] P. T. Wood. Query languages for graph databases. *SIGMOD Record*, 41(1):50–60, 2012.

[9] M. M. Zloof. Query by example. In *AFIPS National Computer Conference*, pages 431–438, 1975.

[1]http://www.transpole.fr/

Flexible Analysis of Plant Genomes in a Database Management System

Sebastian Dorok
Bayer Pharma AG
University of Magdeburg
Germany
sebastian.dorok@ovgu.de

Sebastian Breß
TU Dortmund University
Germany
sebastian.bress@tu-dortmund.de

Jens Teubner
TU Dortmund University
Germany
jens.teubner@tu-dortmund.de

Gunter Saake
University of Magdeburg
Germany
gunter.saake@ovgu.de

ABSTRACT

Analysis of genomes has a wide range of applications from disease susceptibility studies to plant breeding research. For example, different types of barley have differing characteristics regarding draught or salt tolerance. Thus, a typical use case is comparing two plant genomes and try to deduce which genes are responsible for a certain resistance. For this, we need to find differences in large volumes of aligned genome data, which is already available in large genome databases.

The challenge is to efficiently retrieve the genotypes of a certain range of the genome, and then, to determine variants and their impact on the plant organism. State-of-the-art tools are fixed pipelines with a fixed parametrization. However, in practice, users want to interactively analyse genome data and need to customize the parametrization.

In this demonstration, we show how we can support flexible ad-hoc analyses of arbitrary plant genomes using SQL with a small set of user-defined aggregation functions and dynamic parametrization. Furthermore, we demonstrate how genome analysis workflows for variant calling can be applied to our system and provide insights about the performance of our system.

1. INTRODUCTION

The increasing world population also increases the demand for food. Therefore, the harvest of food plants for humans as well as animals must be increased. Typically, plant species are bred to increase harvest or resilience against pests. Genome analysis allows us to determine differences in plant genomes and to determine which genes affect certain traits. Using this knowledge, a more target-oriented breeding is possible.

Genome analysis comprises sequencing of *deoxyribonucleic acid* (DNA) molecules, alignment or assembly of sequenced reads, and analysis of the reconstructed genomes. Typically, a first analysis is *variant calling*, which determines differences between a sample genome and a reference genome. Knowing the differences of a sample genome provides the starting points for further analyses. As plant genomes are huge, e.g., the barley genome comprises 5 billion base pairs, only a step by step analysis is feasible for scientists. Current tools for variant calling allow to call variants on complete genomes or only in specific regions. Thereby, scientists have to know where to look for differences such as the start and end sites of a gene and use fixed tool pipelines that generate the required results. The used tools are mostly command-line driven and flat-file based and put together using scripts. Such setups have the drawbacks that they miss flexibility and are not interactive.

Missing Flexibility The exchange of analysis tools requires knowledge about and adaptions to the scripts and also requires that the tools are compatible regarding used file formats and conventions.

Non-Interactivity When using a script that runs a defined pipeline of tools, the scientist has to wait until the analysis ends and has no opportunity to interrupt the analysis to start another ad-hoc analysis based on intermediate results.

In previous work, we suggested to use main-memory database systems as future genome analysis platform [2]. Moreover, we presented an approach to integrate variant calling into a relational database system [3]. In this demonstration, we present a system that allows users to interactively query and analyse plant genomes using SQL extended with a small set of genome-specfic aggregation functions.

The paper is structured as follows. In Section 2, we present background information on variant calling and analysis methods on genome data. Then, in Section 3, we describe the basic building blocks of our system. The demonstration setup is presented in Section 4. Finally, we conclude in Section 5.

2. BACKGROUND

In this section, we briefly present background information about genome analysis and approaches to integrate genome analysis steps into *database management systems* (DBMSs).

10.5441/002/edbt.2015.45

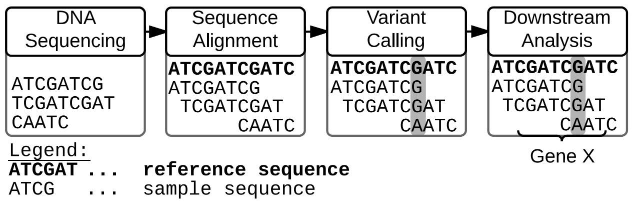


Figure 1: General genome analysis process.

2.1 Genome Analysis

Genome analysis consists of at least four steps. We depict these four steps in Figure 1. In the DNA sequencing step information encoded in DNA molecules is made digitally readable by translating DNA macromolecules into strings of A's, C's,' G's, and T's that encode the nucleo*bases* *A*denine, *C*ytosine, *G*uanine, and *T*hymine, respectively. State-of-the-art DNA sequencers are capable to sequence billions of bases in short time, which increases the amount of genome data to analyse dramatically. Unfortunately, DNA sequencers are not capable to sequence complete DNA molecules at once, but only in small, overlapping pieces. These DNA pieces are called *reads* and have a short length of about 100 base pairs that is rather small compared to the size of a complete genome such as the barley genome with 5 billion base pairs. In order to perform genome analyses, in a second step, DNA molecules must be restored after DNA sequencing from the short reads. After alignment, analysis of genomes starts. Because human or plant genomes are huge, only interesting sites in the genome are analysed at first. To identify such interesting sites, differences between sample genomes and the used reference genome are determined and a variant is called in case a difference is reliable. Such differences can be base exchanges at single genome sites, so called *single nucleotide polymorphisms* (SNPs), or *insertions and deletions* (InDels) of bases regarding the reference genome [6]. At variant sites, further downstream analysis takes place that often requires further data sources to determine the effect of a certain variant. For example, a mutation in a certain barley gene influences the number of spikes of a barley plant [4]. Often, downstream analysis is supported by visualization tools such as IGV that allow scientists to visually inspect and navigate through the genome. In Figure 1, a possible visualization of aligned and annotated genome sequencing data is sketched below the single process steps.

2.2 Related Work

There are several approaches to support genome analysis using database technology. Most of them concentrate on the integration capabilities of DBMSs. For example, Atlas [9] or BioWarehouse [5] integrate heterogeneous data sources using relational database systems and provide specialized APIs to load and access data.

Other approaches aim at integrating genome analysis steps into the database system. Rheinländer et al. present a special join operator that can be used to compute sequence alignments [7]. Wandelt et al. present an approach to perform similarity searches on thousands of genomes [11]. Therefore they use a special index structure that supports fast similarity searches and further allows for impressive compression ratios. In contrast, our approach aims at using minimal-invasive extension mechanisms to provide genome analysis functionality within a relational database system. The approaches can complement each other.

An approach by Rhöm et al. also uses relational databases to primarily store genome data [8]. Therefore, the authors experiment with a special *file wrapper* functionality of SQL Server 2008 to demonstrate how to access data stored in flat files via a DBMS. Additionally, the authors introduce special user-defined functions to manipulate tables and aggregate data items that allow for genome analysis using SQL. Thereby, the authors remark that the performance is problematic due to processing overhead and missing parallelization of user-defined functions. In contrast, our approach is based on main-memory database technology as platform to provide high-performance genome data management. Moreover, we store data directly in the database and use a base-centric database schema to efficiently support genome analysis tasks. Furthermore, we use only user-defined aggregation functions to implement genome-specific analysis tasks.

3. SYSTEM OVERVIEW

In this section, we sketch the basic building blocks of our demonstrator. First, we introduce the database schema used to represent aligned genome data. Then, we present extensions to support genome analysis tasks. Finally, we show how to call SNPs using a custom-aggregation function and how we support dynamic parametrization.

3.1 Database Schema for Aligned Genomes

We use an extended version of the database schema that we presented in previous work for genomes where a reference sequence consists of one contiguous region, such as a single human chromosome or small bacteria genomes [3]. Nevertheless, most animal and plant genomes consist of several contiguous sequences (e.g., chromosomes). This significantly complicates the schema, but is necessary to support variant calling on genomes in general. In Figure 2, we depict the extended entity-relationship schema. We introduced further hierarchies to better represent `reference_genomes`. Not for all genomes a complete reference sequence is known. For example, the reference sequence of barley consists of thousands of known contiguous sequences that have a known order but unknown gaps between them. To represent these contiguous sequences, we introduced the entity `contig`. Moreover, we integrated an entity to represent `sample_genomes`.

An aligned sample genome consists of millions of `reads`, that are aligned to a certain site in a reference genome. Instead of storing the alignment information per read, we split up every aligned read into its single bases and store each base and its mapping to the reference genome separately. In our base-centric database schema, these single bases are represented by the `sample_base` entity. The same idea is used to store the `reference_bases` of a reference genome that consists of contiguous regions instead of reads.

3.2 SQL Genome Extensions

To perform genome analysis tasks using SQL, we have to extend our DBMSs with genome-specific functionality. We use the concept of user-defined aggregation [12] to integrate the required analysis extensions into the DBMS and to make it available via SQL. In combination with our base-centric database schema, we can perform read- or genome-site-specific analyses by grouping and aggregating bases.

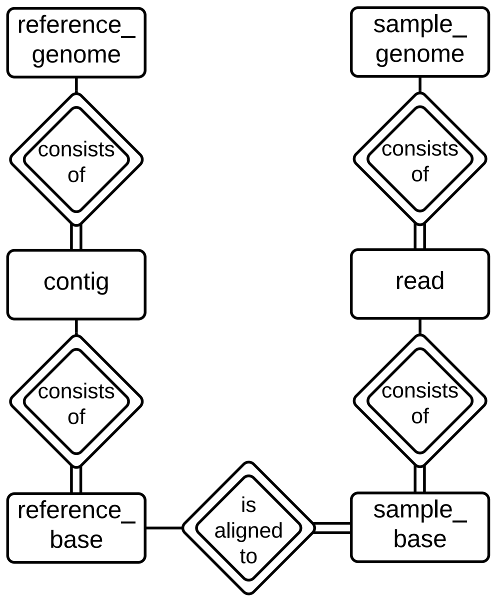


Figure 2: Database schema for aligned genomes showing the association between reference and sample genomes.

In order to call SNPs, we first need to call genotypes. Therefore, we implemented an aggregation function *genotype*, which consumes all bases of a sample genome at a certain position in the genome and determines which genotype is the likeliest at that position. For example, adenine, thymine, cytosine, guanine, or a combination of these bases in case of heterozygous genotypes. We use a frequency based genotype calling algorithm, which is the quasi standard for high coverage genome data [6].

For manual investigation or post processing, it is often necessary that specific reads or the computed genotypes in a certain region can be constructed from the database. Therefore, we propose a user-defined aggregation function *concat_bases*, which concatenates the (computed) bases to reconstruct the sequence, e.g., for the same `read_id`.

Especially, for the analysis of InDels, it is relevant to consider the bases around potential InDels. DNA sequencers have problems reading sequences of bases of the same kind. InDels in such *homopolymer* regions are more likely to be sequencing errors than real variations in the genome. To detect such regions, we integrated a new aggregation function that identifies homopolymer regions.

3.3 SNP Calling

In this section, we explain how to use our database schema and our genome extensions for SQL to call SNPs in a specific genome region.

The goal of SNP calling is to find differences in a sample genome compared to a reference genome, often in coding regions that contain genes. For this, the SNP caller needs to compute the genotype on each position of the aligned sample genome. Our database schema provides simple base-wise access to sample and reference genomes. Therefore, we only have to join the sample_base table with the reference_base table on the reference_id, contig_id, and the position in the reference genome. In order to avoid wrong SNP calls, we exclude reads where the alignment algorithm detected an insertion by filtering sample bases with an insert_offset > 0. Then, we group the sample bases by their position and their respective reference base nd aggregate the sample bases with our aggregation function *genotype*. Finally, we retrieve the variants by comparing the reference base with the computed sample genotype (the called genotype). We illustrate this procedure in an example query in Listing 1.

```
SELECT r.position, r.base as reference_base,
    genotype(s.base) as sample_genotype
FROM sample_base s JOIN reference_base r ON
    r.reference_genome_id = s.reference_genome_id AND
    r.contig_id = s.contig_id AND
    r.position = s.alignment_position
WHERE s.insert_offset = 0
GROUP BY r.position, r.base
HAVING reference_base <> sample_genotype;
```

Listing 1: Query for SNP calling

Based on this query, further analysis queries can be derived. For example, the analyst could reconstruct certain reads in a region of interest, which was discovered by the first query, using our *concat_bases* function. In order to concat bases in a sensible way, we have to guarantee the correct order of bases within a group. Therefore, we use a sort-based grouping approach with the knowledge that all bases within one read reside in the correct order in main memory.

3.4 Dynamic Parametrization

Variant callers typically have many parameters, which customize their behaviour. We support dynamic customization in two ways. First, we can express some parameters (e.g., the consideration of InDels) as simple filter conditions in SQL. Second, the user can set environment variables (e.g., the frequency threshold for the genotype function). For result reproducibility, we include the parametrization as meta data in the query result.

3.5 Putting it all together

With the previously introduced building blocks, users have the ability to analyse genome data in a flexible and interactive way using SQL statements. For example, users can call SNPs using the SQL query presented in Listing 1 and apply additional filter criteria to limit the SNP calling to a genome region of interest. Afterwards, users can check the sequencing coverage in the considered genome region or extract some genotype statistics to validate the variant calling result. Additionally, users can extract the sequences of reads in the genome region of interest. We depict an excerpt of possible analysis tasks and their possible combinations in Figure 3.

4. DEMONSTRATION SETUP

During the demonstration, we will remotely access a server, which runs our system. The server has two Intel Xeon CPUs (E5-2609 v2) @ 2.50GHz and 256GB of main memory @1333 MHz. We implemented our flexible variant calling on plant genomes by integrating our database schema and our aggregation functions in CoGaDB, a column-oriented, GPU accelerated, main-memory DBMS [1]. As evaluation database, we use Harrington barley genome data. We will demonstrate the following aspects in our setup:

Minimal Invasive Extensions and Flexibility: We show the audience how database technology can support

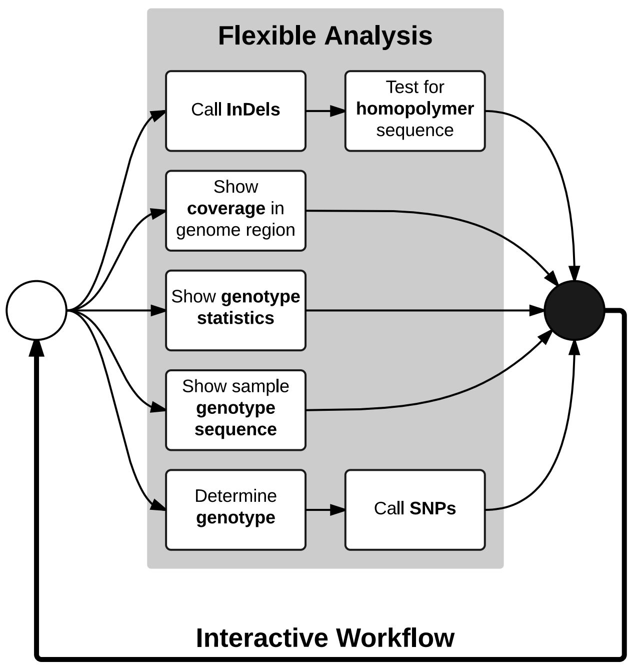


Figure 3: Toolbox for flexible and interactive analysis of genomes.

genome analysis steps such as variant calling. Our intention is to demonstrate that only small extensions of the SQL dialect are sufficient to allow exploration of genome data. We show the flexibility of our system by preparing a list of analysis queries the user can choose from. We assist the user in understanding the analysis queries by annotating each query with information about purpose of the analysis and interpretation of the query result. Additionally, we allow the user to formulate own ad-hoc queries on our database schema.

Simulated Genome Analysis Workflow: The core of the demonstration is to show a typical analysis workflow such as variant calling (c.f. Section 2) that can be performed using our demonstrator. Moreover, the user can experiment with the system on their own to explore the barley genome interactively via SQL. Thereby, our system presents the query results on a SQL commandline interface, but can also function as a backend for other analysis and visualization tools (c.f. Section 2).

Performance of Query Processing: During the interactive analysis, the user can view generated query plans and run-times of single operators, including the specialized genome operators to assess the performance of query processing.

5. CONCLUSION

In this demo, we show how to store aligned genome data in a relational database system and how we can apply real world workflows to our system. Integrating analysis steps into a DBMS and, thus, pushing code to data brings performance advantages due to reduced transfer costs. Moreover, extending SQL with bioinformatics operators combined with improved query performance allows for interactive and ad-hoc querying supporting scientists to prove or disprove hypotheses. This will be demonstrated in the sample workload that we prepare for the demo.

In future work, we want to improve the storage capabilities of our system by applying standard light-weight compression techniques known from database systems such as run-length encoding or dictionary encoding. Moreover, we investigate compression schemes specific for genome data such as referential compression [10] and how to integrate them efficiently into a relational database system. Additionally, we allow the integration of further information such as annotation information.

6. ACKNOWLEDGMENTS

We thank Matthias Lange and Uwe Scholz and their team at the IPK Gatersleben in Germany for fruitful discussions and providing aligned high coverage barley genome data for our experiments.

7. REFERENCES

[1] S. Breß. The Design and Implementation of CoGaDB: A Column-oriented GPU-accelerated DBMS. *Datenbank-Spektrum*, pages 1–11, 2014.

[2] S. Dorok, S. Breß, H. Läpple, and G. Saake. Toward efficient and reliable genome analysis using main-memory database systems. In *SSDBM*, pages 34:1–34:4, 2014.

[3] S. Dorok, S. Breß, and G. Saake. Toward efficient variant calling inside main-memory database systems. In *BIOKDD-DEXA*, 2014.

[4] T. Komatsuda et al. Six-rowed barley originated from a mutation in a homeodomain-leucine zipper I-class homeobox gene. *PNAS*, 104(4):1424–1429, Jan. 2007.

[5] T. J. Lee, Y. Pouliot, V. Wagner, P. Gupta, D. W. J. Stringer-Calvert, J. D. Tenenbaum, and P. D. Karp. BioWarehouse: a bioinformatics database warehouse toolkit. *BMC Bioinformatics*, 7(1):170, 2006.

[6] R. Nielsen, J. S. Paul, A. Albrechtsen, and Y. S. Song. Genotype and SNP calling from next-generation sequencing data. *Nat. Rev. Genet.*, 12(6):443–51, 2011.

[7] A. Rheinländer, M. Knobloch, N. Hochmuth, and U. Leser. Prefix tree indexing for similarity search and similarity joins on genomic data. In *SSDBM*, pages 519–536, 2010.

[8] U. Röhm and J. A. Blakeley. Data management for high-throughput genomics. In *CIDR*, 2009.

[9] S. P. Shah, Y. Huang, T. Xu, M. M. S. Yuen, J. Ling, and B. F. F. Ouellette. Atlas - a data warehouse for integrative bioinformatics. *BMC Bioinformatics*, 6:34, 2005.

[10] S. Wandelt and others. Data Management Challenges in Next Generation Sequencing. *Datenbank-Spektrum*, 12(3):161–171, 2012.

[11] S. Wandelt, J. Starlinger, M. Bux, and U. Leser. RCSI: Scalable Similarity Search in Thousand(s) of Genomes. *PVLDB*, 6(13):1534–1545, 2013.

[12] H. Wang and C. Zaniolo. Using SQL to Build New Aggregates and Extenders for Object- Relational Systems. In *VLDB*, pages 166–175, 2000.

Demonstrating Transfer-Efficient Sample Maintenance on Graphics Cards

Martin Kiefer
Technische Universität Berlin, Germany
kiefer@campus.tu-berlin.de

Max Heimel
Technische Universität Berlin, Germany
max.heimel@tu-berlin.de

Volker Markl
Technische Universität Berlin, Germany
volker.markl@tu-berlin.de

ABSTRACT

Maintaining random data samples under database updates is a fundamental operation in modern database engines. While multiple algorithms exist for this problem, none is tailored to the special case of maintaining data samples on graphics cards. Due to the limited interconnect bandwidth to main memory, any GPU-resident algorithm must try to avoid data transfers across the PCI Express bus where possible – a property that we call transfer-efficient. In this demonstration, we present an approximate, transfer-efficient sample maintenance algorithm that piggybacks on a GPU-accelerated selectivity estimator and utilizes query feedback to selectively identify and replace outdated points. We provide an implementation of the algorithm and interactively demonstrate its quality and its transfer performance in comparison to traditional maintenance algorithms.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems

1. INTRODUCTION & MOTIVATION

Collecting and maintaining data samples is a fundamental operation in a modern database engine. One of the main advantages of algorithms that can operate on samples lies in their ability to trade-off performance against result accuracy by reducing – or increasing – the sample size. This property allows us to efficiently mask limited resources, as long as our application can tolerate the loss of result accuracy. Examples of such "tolerant" applications include selectivity estimation [14, 15], approximate query processing [3, 4], data mining algorithms [17], interactive data exploration, and data visualization.

One area where sampling-based methods are of particular interest are GPU-accelerated databases. The usability of graphics cards for data-intensive operations is severely limited by two bottlenecks: The scarce availability of on-card device memory and the slow data transfer speeds from main memory across the PCI Express bus [6]. We can avoid both bottlenecks by keeping a fixed number of sampled data points on the device to quickly compute approximate results from. As long as the underlying database remains static, this approach works very well and does not require any further transfers across the PCI Express bus.

Sadly, in the real world, datasets seldomly remain static. In order to stay representative, all changes that are applied to the underlying database have to be mirrored to the sample as well. This so-called *sample maintenance* problem is well-known, and multiple algorithms exist for it [8, 9, 16, 18]. However, while maintenance algorithms guarantee to keep the sample representative, they usually require us to replay database updates. In case of a GPU-resident data sample, this means that we need to copy all updates across the PCI Express bus. These additional transfers restrict the available PCI Express bandwidth, leading to performance penalties for "actual" data processing applications running on the GPU. Ideally, we want a *transfer-efficient* maintenance algorithm that only transfers data if it is absolutely necessary. In this paper, we are discussing a possible sample maintenance algorithm that aims for this property. Our primary contributions are:

1. We introduce an approximate, but transfer-efficient maintenance algorithm for samples on graphics cards. Our algorithm piggybacks on a GPU-accelerated selectivity estimator and utilizes query feedback to track outdated points. This approach allows us to selectively replace outdated points in the sample without having to mirror all updates to the graphics card.

2. We provide an implementation of our algorithm integrated into the open-source relational database engine PostgreSQL[1].

3. We interactively demonstrate the performance of our algorithm in comparison to other replacement strategies with regard to both sample quality and the required data transfers across the PCI Express bus.

2. GPU-BASED SAMPLE MAINTENANCE

Assume a d-dimensional relation R with cardinality $|R|$ that is stored within a "regular" relational database system. From R, we collect a fixed-size random sample $S \subseteq R$ and push it to the graphics card. The sample size $|S|$ is fixed and chosen in advance to a) provide sufficient confidence for the

[1]The source code is available at: `goo.gl/aQSQNd`.

approximated results, and to b) fit within the limited device memory on the graphics card. Maintaining such a GPU-resident sample when database updates occur on the host is an interesting problem that, to the best of our knowledge, has not been discussed in the literature so far.

The simplest maintenance scenario is an insertion-only workload. In this case, *Reservoir Sampling* [18] is the ideal choice: It pushes newly inserted points to the sample with probability $|S| \,/\, |R|$, replacing a random sample point. From a transfer-efficiency perspective, Reservoir Sampling is optimal: We only push exactly those data items to the graphics card that end up in the sample.

The general case of mixed workloads containing insertions, updates and deletions is more interesting: General maintenance algorithms usually handle insertions similar to Reservoir Sampling, but have special rules to deal with updates and deletions [8, 9, 16]. Take for example the *Correlated Acceptance Rejection Sampling algorithm* (CAR) [16]: When a new point $\vec{t}$ is inserted into R, a random number n is drawn from the binomial distribution $BINOM(|S|, 1/|R|)$ and n random sample points are replaced by an instance of $\vec{t}$. When a deletion occurs, all instances of the deleted point are removed from the sample and exchanged by points drawn uniformly from R. Updates are handled by directly applying them on the sample. This means that, while insertions incur maintenance costs of $\mathcal{O}(1)$, deletions and updates require additional costs of $\mathcal{O}(|S|)$.

A straightforward way to adjust algorithms like CAR for GPU-resident samples would be to maintain a sample copy on the host, apply the maintenance algorithm there, and then mirror sample updates to the GPU memory. While this approach would indeed be transfer-efficient, we still had to pay the $\mathcal{O}(|S|)$ maintenance costs for every update and deletion. For large sample sizes, these additional costs can become quite significant and slow down the system, which is why we want to push as much of the maintenance work as possible directly to the faster graphics card.

However, running existing sample maintenance algorithms on the graphics card requires us to mirror every deletion and update across the PCI Express bus, even if they do not apply to any points in the sample. For instance: Running CAR on the graphics cards incurs a sequence of two mandatory transfers for each deletion: One to transfer the deleted item, and one to reply with a list – or bitmap – identifying all deleted tuples, so that the database can sample a sufficient amount of tuples and transfer them to the correct positions in the GPU memory. These additional transfers across the limited PCI Express bus might easily become a problem: Even if they do not fully block the bus, they will take a significant chunk of the available bandwidth away from other GPU-resident applications. This is especially relevant when keeping in mind that transfers below a minimum length (on the order of a few Kilobytes) do not achieve maximum throughput [7].

3. BACKGROUND: CALCULATING KERNEL DENSITY ESTIMATORS ON GPUS

We investigated the sample maintenance problem in the context of a GPU-accelerated selectivity estimator [11]. In order to convey the necessary background knowledge, we will now give a brief introduction into this topic.

Given a relation R with attributes $(A_1, ..., A_d)$ and an arbitrary query region $\Omega \subseteq D_1 \times ... \times D_d$, selectivity estimators approximate the fraction $\frac{|\sigma_{\vec{x}\in\Omega}(R)|}{|R|}$ of tuples qualifying the query. In our case, we assume that the attributes are from the domain of real numbers.

Multiple authors have proposed using *Kernel Density Estimators* (KDEs) to approach this task [5, 10, 11]. The principle idea behind KDEs is visualized in Figure 1: Based on a sample (Figure 1b) drawn from R (Figure 1a), KDE places local probability density functions – the so-called *kernels* – around the sample points (Figure 1c). The probability density function for the overall data is then estimated by summing and averaging over those kernels (Figure 1d).

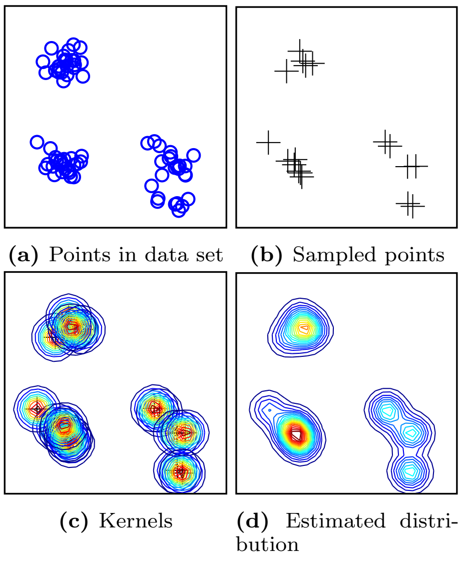

(a) Points in data set **(b)** Sampled points **(c)** Kernels **(d)** Estimated distribution

Figure 1: A Kernel Density Estimator approximates the underlying distribution of a given dataset (a) by picking a random sample (b), centering local probability distributions (kernels) around them (c), and averaging the local distributions (d).

Formally, given a sample $S = \{\vec{t}^{(1)}, \vec{t}^{(2)}, ..., \vec{t}^{(s)}\} \subseteq R$, the Kernel density Estimator $\hat{p}_H(\vec{x}) : \mathbb{R}^d \rightarrow \mathbb{R}$ is defined as:

$$\begin{aligned} \hat{p}_H(\vec{x}) &= \tfrac{1}{s}\textstyle\sum_{i=1}^{s} K_H(\vec{t}^{(i)} - \vec{x}) \\ &= \tfrac{1}{s\cdot|H|}\textstyle\sum_{i=1}^{s} K(H^{-1}[\vec{t}^{(i)} - \vec{x}]) \end{aligned} \quad (1)$$

Here, $K : \mathbb{R}^d \rightarrow \mathbb{R}$ denotes the *kernel function*, which defines the shape of the local probability density functions. Typical choices are Gaussian – a multivariate standard normal distribution –, or Epanechnikov, which is a truncated second-degree polynomial. $H \in \mathbb{R}^{d\times d}$ is the *bandwidth matrix*, which controls the spread of the kernel function. Picking the optimal bandwidth is a difficult problem that is out of the scope of this demonstration. We assume that it is selected by a data-driven bandwidth optimizer [1].

We can now predict the selectivity for a (rectangular) query region Ω by integrating $\hat{p}_H(\vec{x})$ over all points in the region:

$$\hat{p}_H(\Omega) = \int_\Omega \hat{p}_H(\vec{x})d\vec{x} = \frac{1}{s}\sum_{i=1}^{s} \underbrace{\int_\Omega \frac{K(H^{-1}[\vec{t}^{(i)} - \vec{x}])}{|H|}}_{\hat{p}_H^{(i)}(\Omega)} \quad (2)$$

This equation can be efficiently evaluated in parallel: First, each thread independently computes the individual probability contribution $\hat{p}_H^{(i)}(\Omega)$ for a single sample point $\vec{t}^{(i)}$. The estimate is then computed as the averaged sum over all individual contributions – which can be efficiently computed via a parallelized binary reduction scheme [13]. For further details on KDEs, and on how we designed a GPU-accelerated selectivity estimator based on them, we kindly refer to our publication [11].

4. INTRODUCING THE KARMA METRIC

We now introduce a novel approach for sample maintenance in the context of a GPU-based Kernel Density Estimator that is used for selectivity estimation. Our approach is based on *query feedback*: After the execution of a query covering region Ω, the true selectivity $p(\Omega)$ of the region is known. The principle idea behind query feedback methods is to utilize this additional information to incrementally adjust the estimation model [2].

In particular, we can compute for each sample point $\vec{t}^{(i)}$ the impact of its probability contribution $\hat{p}_H^{(i)}(\Omega)$ on the (absolute) estimation error $\mathcal{L}_{abs}\left(p\left(\Omega\right), \hat{p}_H\left(\Omega\right)\right)$. For this, we first calculate the adjusted estimate $\hat{p}_H^{-(i)}(\Omega)$ by removing the point's contribution from the estimate $\hat{p}_H(\Omega)$:

$$\hat{p}_H^{-(i)}(\Omega) = \frac{\hat{p}_H(\Omega) \cdot s - \hat{p}_H^{(i)}(\Omega)}{s-1} \tag{3}$$

Now, $\hat{p}_H^{-(i)}(\Omega)$ is simply the selectivity that our estimator would have predicted if point $\vec{t}^{(i)}$ had not been part of the sample. Based on this, we can compute the adjusted estimation error $\mathcal{L}_{abs}\left(p\left(\Omega\right), \hat{p}_H^{-(i)}\left(\Omega\right)\right)$, which is the estimation error if $\vec{t}^{(i)}$ had been removed. The principle idea behind our maintenance algorithm is then simple: A sample point that significantly reduces the estimation error by its absence is likely misrepresenting the distribution in the data set and should be replaced. Accordingly, we define the *Karma* $K^{(i)}(\Omega)$ for sample point $\vec{t}^{(i)}$ as its impact on the estimation error:

$$K^{(i)}(\Omega) = s \cdot \Big(\mathcal{L}_{abs}(p\left(\Omega\right), \hat{p}_H(\Omega)) - \mathcal{L}_{abs}\Big(p\left(\Omega\right), \hat{p}_H^{-(i)}(\Omega)\Big)\Big) \tag{4}$$

We multiply by the sample size s to normalize the values to $[-1, 1]$: Karma values close to one correspond to sample points that significantly improved the estimation quality for query region Ω, while negative values are associated with points that had a negative impact. If the selectivity is overestimated, points contributing to the overestimation will be penalized, while points outside of the query region – or those without significant contributions – will be rewarded, vice-versa for underestimated selectivities.

By aggregating Karma values over a sequence of query regions $[\Omega_1, ..., \Omega_n]$, we obtain an indicator for the contribution of sample points over multiple queries. Accordingly, we recursively define our notion of *cumulative Karma* via the following recursion:

$$K^{(i)}([\Omega_1, ..., \Omega_n]) = \begin{cases} \alpha \cdot K^{(i)}(\Omega_n) + \\ \quad (1-\alpha) \cdot K^{(i)}([\Omega_1, ..., \Omega_{n-1}]) & n > 1 \\ \alpha \cdot K^{(i)}(\Omega_n) & n = 1 \end{cases} \tag{5}$$

In this equation, $\alpha \in (0, 1]$ is a constant factor for applying exponential smoothing to limit the influence of historic Karma values. This approach helps us to achieve faster reactivity on changing data, as well as to improve the method's robustness with respect to outliers.

The cumulative Karma is used as the foundation for our heuristic sample maintenance with focus on reestablishing good estimation results. This is done by resampling points with large negative Karma values as they are likely to misrepresent the true data distribution. This approach relieves us from mirroring all changes to the sample. $K^{(i)}$ can be computed easily on top of our KDE-based estimator: The calculation can be performed by executing one additional embarrassingly parallel computation on the GPU and allocating an additional field for the most recent values of $K^{(i)}$ for all points in the sample.

Note that this algorithm does not provide true sample uniformity, but instead aims at maintaining a sample that fits the underlying distribution in the queried regions.

5. DEMONSTRATION

In our demonstration, we first introduce our implementation in PostgreSQL and give an overview of the modifications that were applied. Afterwards we provide an interactive graphical evaluation of several sample maintenance algorithms under variable parameters and workload characteristics.

5.1 System Overview

All presented algorithms were integrated into the open-source database PostgreSQL 9.3.1. The GPU-accelerated algorithms were implemented in a hardware-oblivious way using OpenCL, which allows us to operate on all devices supporting the standard – including graphics cards, and multi-core CPUs [12]. We provide PostgreSQL control variables to control the KDE-based selectivity estimation for selected tables and to select the sample maintenance method. Besides integrating the use of KDEs in the estimator for qualifying queries, we added hooks after query executions, insertions and deletions, which are used to call the selected sample maintenance algorithms.

5.2 Compared Methods

We compare the following sample maintenance algorithms during our interactive presentation:

No maintenance (NONE) is our first baseline. In this method, we do not perform any sample maintenance, demonstrating the severity of estimation error degradation as updates are applied to the database.

Correlated Acceptance Rejection Sampling (CAR) is used as a baseline for existing maintenance algorithms and is implemented as explained in Section 2.

Periodic Random Replacement (PRR) is our third baseline. It replaces a random item from the sample with a newly sampled item every n changes to the base data.

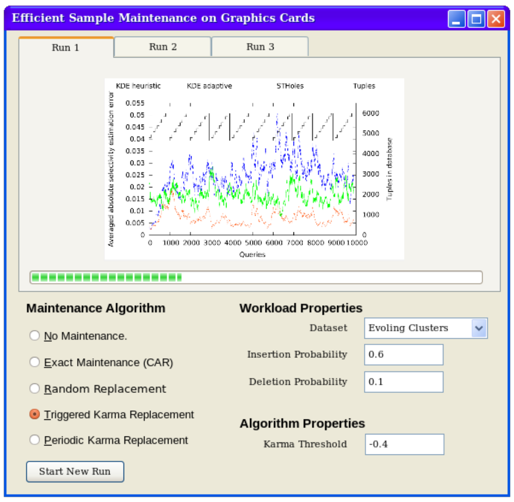
Triggered Karma Replacement (TKR) replaces sample points when their cumulative Karma goes below a given threshold γ. A bitmap identifying points that will be resampled has to be calculated after Karma calculation and is transferred to the CPU to trigger resampling.

Periodic Karma Replacement (PKR) periodically replaces the sample point with the worst cumulative Karma every m queries. The sample point $\vec{t}^{(i)}$ with the minimum Karma is efficiently identified on the graphics card via a parallel reduction scheme [13].

5.3 Demonstration Overview

At the beginning of the demonstration, the user can choose from pre-selected dataset choices, each with different properties and sizes. We then collect and transfer a new sample for the selected dataset to the graphics. Afterwards, the user is prompted to select and configure a maintenance algorithm, and to specify characteristics of the query workload (e.g. the probability of insertions and deletions).

After starting the experiment, we continuously run random selection queries, plotting the average selectivity estimation error, as well as the number of transfered tuples that were required for the maintenance algorithms. This allows the user to inspect in real-time how the sample quality, and the required data transfers develop. The presentation will be delivered with an interface similar to Figure 2.



Maintenance Algorithm

- No Maintenance.
- Exact Maintenance (CAR)
- Random Replacement
- [✓ Triggered Karma Replacement]
- Periodic Karma Replacement

Workload Properties

Dataset: Evoling Clusters
Insertion Probability: 0.6
Deletion Probability: 0.1

Algorithm Properties

Karma Threshold: -0.4

Figure 2: Overview of the demonstration interface: The user can select the desired sample maintenance method and specify algorithm and workload properties. When the user starts the configured experiment, we plot the average estimation-error from the sample, the required transfers across the PCI Express bus, and the cumulative updates to the database.

6. REFERENCES

[1] *Multivariate Density Estimation - Theory, Practice and Visualization.* John Wiley & Sons, Inc., 1992.

[2] A. Aboulnaga and S. Chaudhuri. Self-tuning histograms: Building histograms without looking at data. In *ACM SIGMOD Record*, volume 28, pages 181–192. ACM, 1999.

[3] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 29–42. ACM, 2013.

[4] B. Babcock, S. Chaudhuri, and G. Das. Dynamic sample selection for approximate query processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 539–550. ACM, 2003.

[5] B. Blohsfeld, D. Korus, and B. Seeger. A comparison of selectivity estimators for range queries on metric attributes. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, pages 239–250, New York, NY, USA, 1999. ACM.

[6] S. Breß, M. Heimel, N. Siegmund, L. Bellatreche, and G. Saake. Gpu-accelerated database systems: Survey and open challenges. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XV*, pages 1–35. Springer, 2014.

[7] Y. Fujii, T. Azumi, N. Nishio, S. Kato, and M. Edahiro. Data transfer matters for gpu computing. In *Parallel and Distributed Systems (ICPADS), 2013 International Conference on*, pages 275–282, Dec 2013.

[8] R. Gemulla, W. Lehner, and P. J. Haas. A dip in the reservoir: Maintaining sample synopses of evolving datasets. In *Proceedings of the 32nd international conference on Very large data bases*, pages 595–606. VLDB Endowment, 2006.

[9] P. B. Gibbons, Y. Matias, and V. Poosala. Maintaining a random sample of a relation in a database in the presence of updates to the relation, Jan. 4 2000. US Patent 6,012,064.

[10] D. Gunopulos, G. Kollios, J. Tsotras, and C. Domeniconi. Selectivity estimators for multidimensional range queries over real attributes. *The VLDB Journal*, 14(2):137–154, Apr. 2005.

[11] M. Heimel and V. Markl. A first step towards gpu-assisted query optimization. In *ADMS@ VLDB*, pages 33–44. Citeseer, 2012.

[12] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl. Hardware-oblivious parallelism for in-memory column-stores. *Proceedings of the VLDB Endowment*, 6(9):709–720, 2013.

[13] D. Horn. Stream reduction operations for gpgpu applications. *Gpu gems*, 2:573–589, 2005.

[14] P.-A. Larson, W. Lehner, J. Zhou, and P. Zabback. Cardinality estimation using sample views with quality assurance. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 175–186. ACM, 2007.

[15] R. J. Lipton, J. F. Naughton, and D. A. Schneider. *Practical selectivity estimation through adaptive sampling*, volume 19. ACM, 1990.

[16] F. Olken and D. Rotem. Maintenance of materialized views of sampling queries. In *Data Engineering, 1992. Proceedings. Eighth International Conference on*, pages 632–641, Feb 1992.

[17] H. Toivonen et al. Sampling large databases for association rules. In *VLDB*, volume 96, pages 134–145, 1996.

[18] J. S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985.

NoFTL for Real: Databases on Real Native Flash Storage

Sergey Hardock #1, Ilia Petrov #2, Robert Gottstein #1, Alejandro Buchmann #1

#1 *Databases and Distributed Systems Group TU-Darmstadt*, #2 *Data Management Lab, Reutlingen University*

#1 `{hardock | gottstein | buchmann}@dvs.tu-darmstadt.de`, #2 `ilia.petrov@reutlingen-university.de`

ABSTRACT

Flash SSDs are omnipresent as database storage. HDD replacement is seamless since Flash SSDs implement the same legacy hardware and software interfaces to enable backward compatibility. Yet, the price paid is high as backward compatibility masks the native behaviour, incurs significant complexity and decreases I/O performance, making it non-robust and unpredictable. Flash SSDs are black-boxes. Although DBMS have ample mechanisms to control hardware directly and utilize the performance potential of Flash memory, the legacy interfaces and black-box architecture of Flash devices prevent them from doing so.

In this paper we demonstrate NoFTL, an approach that enables native Flash access and integrates parts of the Flash-management functionality into the DBMS yielding significant performance increase and simplification of the I/O stack. NoFTL is implemented on real hardware based on the OpenSSD research platform. The contributions of this paper include: (i) a description of the NoFTL native Flash storage architecture; (ii) its integration in Shore-MT and (iii) performance evaluation of NoFTL on a real Flash SSD and on an on-line data-driven Flash emulator under TPC-B,C,E and H workloads. The performance evaluation results indicate an improvement of at least 2.4x on real hardware over conventional Flash storage; as well as better utilisation of native Flash parallelism.

1. INTRODUCTION

Many basic database architectural principles and algorithms have been designed around the properties of HDD. Flash memories provide a set of different I/O characteristics and promise to speedup the critical I/O path. We argue that the design of the storage architecture is not well suited for new kinds of memory in terms of both software and hardware. Flash devices still support the same block level interface as HDDs, which ensures backwards compatibility and eases replacement, but is also a major source of unpredictability and non-robustness.

The low-level block interface compatibility is realized by the *Flash Translation Layer (FTL)* that is executed inside the storage device on top of limited hardware. The FTL creates a black-box around the Flash memory, masking its performance characteristics and emulating a HDD-like behaviour [5, 3]. The FTL yields: (i) significant overhead; (ii) unpredictable and state-dependent performance due to background processes [5, 4]; (iii) inability to optimize the DBMS I/O behaviour to new kinds of storage[4, 10]; and last but not least, uncures (iv) high costs of Flash SSDs.

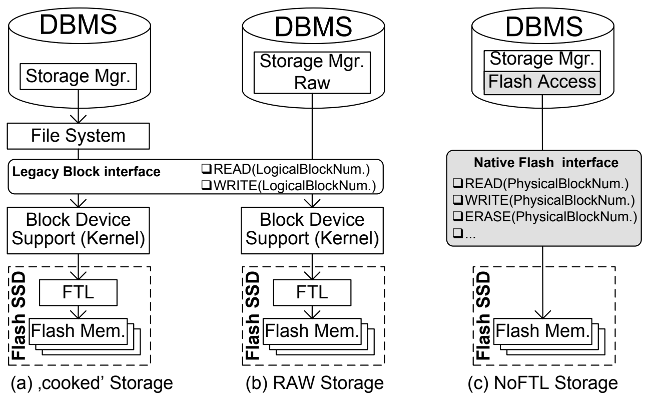


Figure 1: DBMS storage alternatives

Historically, database systems assume direct control over the hardware and the I/O stack to increase performance. Traditionally a DBMS would use a file system based ("cooked") storage on traditional block devices (Figure 1.a). Database systems on raw storage (Figure 1.b) eliminate file system overhead, enable raw storage access and direct physical data placement, achieving better performance [14]. Newer approaches propose a departure from block device interfaces, achieving: atomic writes, computational efficiency and parallelism [15], stripped down FTL and a native interface to host [4, 10]. With *NoFTL* (Figure 1.c) we consider native Flash access, and explore approaches to natural integration of FTL functionality in the DBMS.

Contributions. *NoFTL* removes all intermediate abstraction layers along the critical I/O path (block device interface, file system and FTL), and enables the DBMS to control the Flash memory directly. This minimizes the overhead of garbage collection (GC) and wear-leveling (WL), allowing the DBMS to efficiently utilize the Flash memory. The contributions of this paper are: (i) we implemented NoFTL on real hardware based on the OpenSSD research platform as well as on a real-time data-driven Flash emula-

10.5441/002/edbt.2015.47

tor; the latter was validated against OpenSSD and extended to support parallelism; (ii) we incorporated different FTLs (DFTL, Faster); (iii) live TPC-C, -B and -H tests under Shore-MT indicate a *NoFTL* performance improvement of 1.5x to 2.4x. NoFTL has been initially demonstrated in [8]. In contrast to [8] the current demonstration has the following improvements: (a) the real time emulator has been extended to handle parallelism (Section 3.2); (b) NoFTL has been implemented on the hardware research platform - OpenSSD (Section 3.3); (c) the database integration is deepened.

2. RELATED WORK

Numerous designs of FTLs that can be classified as Page-Block- or Hybrid-/Log-Block- Mapping FTLs have been proposed ([16], [7], [11], [12] etc.). An evaluation and comparison of different FTLs is provided in [5, 6, 13]. DFTL is introduced in [7] as a page-mapping FTL. There are multiple Flash simulation frameworks such as FlashSim [9] or DiskSim. There is ongoing research on omitting certain on-device FTL functionalities: [15] is not using the block I/O interface; [4] presents a hybrid approach which can bypass the on-device FTL. Specialized Flash Server Storage moves the FTL from a device into the driver, such as FusionIO [1]. NoFTL completely removes the on-device FTL, enabling the DBMS to take full control over the Flash device. An earlier demonstration of NoFTL is provided in [8], while here we: i) extend the emulator with parallelism; ii) port and present NoFTL on real hardware - the OpenSSD board; and iii) further integrate NoFTL into the DBMS.

3. THE NOFTL CONCEPT

Due to the black-box design of modern SSDs neither (i) the information about internal Flash architecture can be utilized by data placement algorithms in the DBMS; nor (ii) the DBMS status information about stored data and I/O (runtime & history) can be used to optimize the FTL. Furthermore, the DBMS can experience significant fluctuations in I/O latency and throughput that are state-dependent and result from expensive FTL operations (e.g., WL and GC). For instance, the average 4KB random write latency on a SLC SSD is 0.450ms, while frequent FTL-specific outliers under heavy load can reach 80ms [5]. In the same time, the efficiency of the FTL maintenance functionality is strongly coupled to limited on-device computational resources (e.g., single ASIC controller and up to 512MB of RAM).

NoFTL is in an attempt to overcome the aforementioned disadvantages. *Under NoFTL the DBMS operates directly on native Flash memory*, without intermediate layers such as file system, block-device layer or on-device FTL (Figure 1). The Flash maintenance (address translation, GC, WL, etc.) is integrated into the DBMS. Such an integration is based on the following important observation: *large parts of the FTL naturally leverage the functionality of existing DBMS modules such as the storage manager, the free space manager or the transaction manager* (Figure 2).

The general integration strategy is the optimization of Flash maintenance and DBMS algorithms based on the: (i) usage of more powerful computational and memory resources of the host system (e.g., address mapping); (ii) usage of the DBMS run-time information and knowledge about the stored data and I/O (e.g., WL & GC); (iii) elimination of redundant functionality along the I/O path (e.g., buffer management, free space management and address mapping in file system and FTL); (iv) optimisation of DBMS data placement and access algorithms based on the Flash layout (e.g., assignment of DBMS background flushers to physical address space regions).

Noteworthy is that *under NoFTL the DBMS is not confronted with the intricate low-level NAND control.* The Flash SSD is still assumed to have a thin hardware management layer (Figure 2) providing low-level NAND chip management, such as timing and synchronisation, low-level row and column address translations, channel and bus management. The functionality of the controller can optionally be implemented as a kernel driver (cheap but slow).

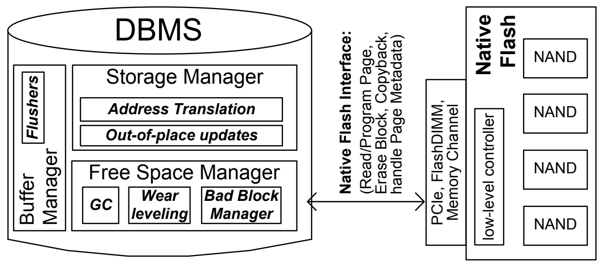


Figure 2: General Architecture of NoFTL

The minimal set of commands that the *native Flash interface* provides is: PAGE_READ and PAGE_PROGRAM with data transfer; COPYBACK_PROGRAM and BLOCK_ERASE without transfer of user data. Meaningful are also the variants of those commands to support reading or writing the series of pages (not necessary logically adjacent), which would be further translated into appropriate optimized commands according to the Flash specification (like READ/PROGRAM _CACHE_RANDOM/SEQUENTIAL in ONFI NAND). The protocol must also include the identification command (similar to HDIO_GETGEO for HDDs), which allows the DBMS to receive detailed information about the architecture of the Flash SSD (e.g., channels, LUNs, Flash type, etc.).

3.1 Host Memory Resources

The logical-to-physical address translation is the core component of an out-of-place update strategy, and is one of the most memory consuming subsystems within modern Flash SSDs. Since the amount of on-device memory is insufficient to hold a complete mapping table at page-level granularity, multiple alternatives were proposed in the recent years. Three of them, recognized as state-of-the-art, are page-level FTLs DFTL (demand-based FTL) [7] and LazyFTL [12], as well as the hybrid mapping scheme FASTer [11].

DFTL and LazyFTL keep mapping information at page-level granularity, but only a small fraction of it is cached. They introduce computational (maintenance and look-ups of cached mappings) and I/O overheads (page-ins and -outs to fetch from and store mappings on Flash). Our earlier results [8] indicate a performance slowdown of DFTL over pure page-level mapping (where the whole mapping table is cached) of up to 3.7x under TPC-C and -B benchmarks. In FASTer the larger part of Flash memory is mapped at block-level granularity (data block area), while only a small part (log block area) uses page-level mappings. All updates and write requests are first performed in the log block area, and as soon as free space in that region runs out those updates are merged with the corresponding blocks in the data block area. Merges result in expensive additional background I/Os (Figure 3). For TPC-B, -C and -E the garbage collection overhead in FASTer is almost twice as high as in NoFTL (Figure 3). This overhead negatively influences the foreground performance. Solely the use of DBMS-integrated page-level mapping in NoFTL results in 2.4x and 2.25x improvement in transactional throughput (TPS) for TPC-C and -B, respectively. The high write amplification in FASTer significantly reduces the longevity of the Flash SSD.

IO type	TPC-C SF=30		TPC-B SF=350		TPC-E 1K Customers	
	Absolute	Relative	Absolute	Relative	Absolute	Relative
COPYBACK	16 465 930	1.98x	17 295 713	2.15x	1 805 540	1.97x
ERASE	129 317	1.73x	135 839	1.82x	14 231	1.68x
Off-line trace-driven testing. Traces were recorded on in-memory database running the benchmarks for 60 minutes.						

Figure 3: Absolute and relative I/O overhead of garbage collection under FASTer and NoFTL.

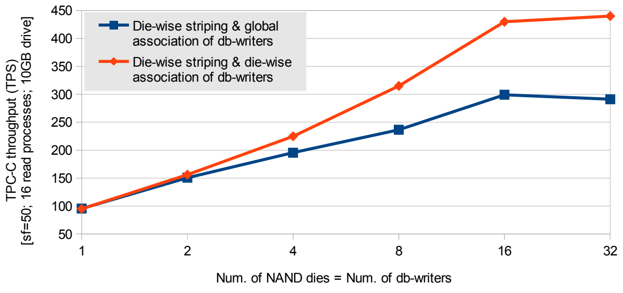
3.2 Utilization of Flash parallelism

Many DBMS algorithms can be optimized based on the architecture of underlying Flash SSDs. Direct control over physical data placement allows to efficiently utilize native Flash parallelism. For instance, SATA2 allows for at most 32 concurrent I/O commands; whereas a commodity Flash SSD with 8 to 10 chips is able to execute up to 160 concurrent I/Os (8-16 commands/chip). To make better use of the available Flash parallelism, we have incorporated the knowledge about Flash architecture into the logic of database writer processes (db-writers). The basic idea is to remove the contention for physical resources among db-writers. Instead of having multiple db-writers, where each is responsible for a subset of dirty pages from the whole address space, we have assigned each db-writer to a certain physical region (i.e., set of NAND chips). Therefore, each db-writer receives a distinct subset of dirty pages that belongs to a corresponding physical address space, and does not compete for physical storage with db-writers assigned to other regions. Depending on the workload and the size of the regions it is also possible to assign several db-writers to a single region. We have implemented this optimization in Shore-MT. We can demonstrate an improvement of throughput over the initial implementation with the same number of background writer processes of up to 1.5x for TPC-C and 1.43x for TPC-B benchmarks (see Figure 4). With an increasing amount of Flash parallelism and more db-writers (leveraging the parallelism), the difference in the transactional throughput increases. In the standard approach with a global assignment the response time for each single db-writer increases, due to the higher contention for Flash chips.

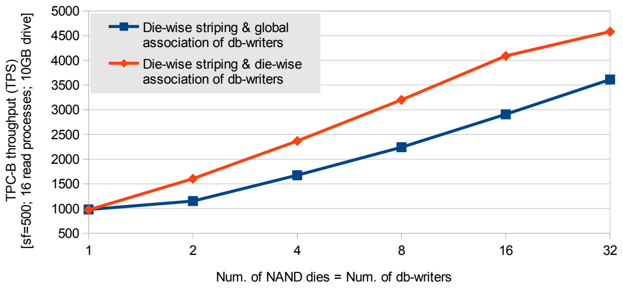
3.3 NoFTL Testbed

We have implemented the NoFTL concept in Shore-MT [2], which is a recognised open-source storage engine supporting ACID transactions, ARIES-type logging, Indices, Buffer management. Furthermore, Shore-MT supports raw devices and standard TPC benchmark implementations. The NoFTL-version of the storage engine was evaluated on the real hardware OpenSSD board, as well as on our enhanced version of the real-time Flash emulator.

OpenSSD board. The OpenSSD project aims to provide an open hardware Flash research platform (see Figure 5). It allows to program the firmware running on the on-



(a) TPC-C



(b) TPC-B

Figure 4: Tx. throughput of TPC-C/-B with global and Flash-aware assignment of db-writers.

device controller to a certain extent, and to test different FTL schemes and algorithms. The board contains a set of operational FTLs, among them the popular FASTer scheme [11]. To make OpenSSD perform as a native Flash board we have removed the FTLs and modified the I/O protocol to support the native Flash (ATA Pass-Through, Section 3).

Real-time Flash emulator. We have enhanced our real-time Flash emulator [8] to support complex highly parallel Flash architectures. The emulator is implemented as a device driver in the Linux kernel. The usage of low-level kernel primitives guarantees high precision ($\sim 1\mu s$) in simulation of I/O latencies. There is no loss of accuracy with increasing capacity of the emulated drive, however, the latter is limited by the available RAM resources of the host system. The emulated Flash storage provides either a block-device interface, emulating a SSD or a character device interface, emulating native Flash. The emulator allows to investigate parallelism, different Flash layouts or NAND types (SLC,MLC,TLC) as well as characteristics such as wear, which is not possible with OpenSSD. The emulator's behavior and characteristics have been validated against the OpenSSD platform.

4. DEMONSTRATION

The demonstration is performed based on two platforms: the OpenSSD hardware research platform and a real time Flash emulator. We compare NoFTL against the conventional DBMS storage, based on black-box Flash SSDs (Figure 1.a, 1.b). For the latter scenario (Figure 6.a) we choose two state-of-the-art FTL as counterparts: (i) DFTL [7] (page-level mapping); and (ii) FASTer [11] (hybrid mapping). All demonstration scenarios are performed live either on real hardware (OpenSSD board), or on the Flash emulator.

Demo Scenario 1 - Validation of Flash emulator.

In this scenario we stress the emulator with the Linux FIO tool to showcase: (1) Its accuracy and reconfigurability, i.e., test different internal architectures of Flash memory on synthetic benchmarks; and (2) Investigate the DBMS

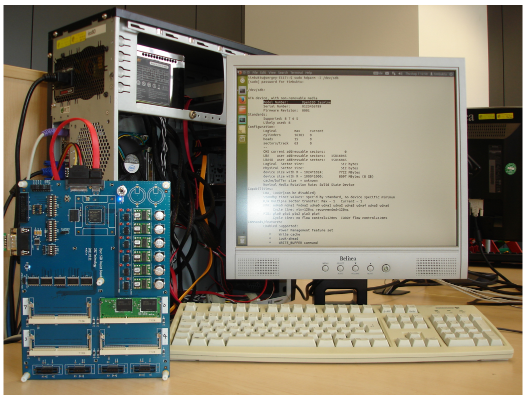


Figure 5: OpenSSD connected to the test-bed.

utilisation of richer parallelism under NoFTL to improve transactional throughput. We also perform the live validation of the Flash emulator against the OpenSSD hardware by configuring the former with the properties and architecture of latter and comparing the performance results of live TPC benchmarks. For each demonstration run the audience is presented the resulting diagrams for transactional throughput and response time as well as statistics regarding erasures, writes and garbage collection activity.

Demo Scenario 2 - DBMS performance under NoFTL.

The general architecture of the NoFTL demonstration testbed is depicted in Figure 6. During the demonstration the audience can select any of the TPC benchmarks (-H, -B, -C or -E) and a demonstration platform (OpenSSD or Flash emulator). Furthermore, the audience can configure the Flash layout as well as the number of DBMS flushers to experience the influence of the different strategies. With an increasing amount of Flash parallelism and more db-writers (leveraging the parallelism), the difference in the transactional throughput increases. Test results comprise Shore-MT's output, intermediate and average transactional throughput, as well as detailed statistics of I/O operations and GC overhead. Furthermore, we demonstrate the influence of the improved ("Flash-aware") allocation and assignment strategy of background writers in Shore-MT (Sect. 3.2).

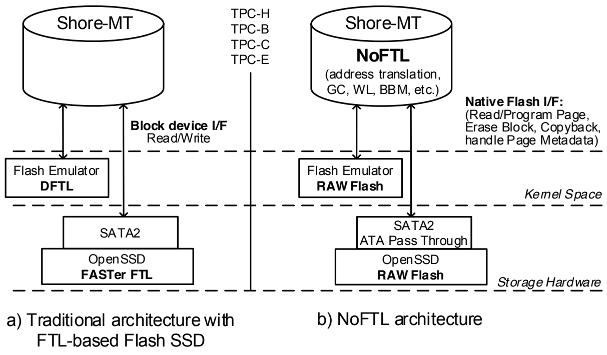


Figure 6: Demonstration scenarios.

5. CONCLUSIONS

We demonstrate *NoFTL* - an approach that yields a significant simplification of the I/O stack; integrates Flash management in the DBMS; allows direct access to storage and exposure of a native Flash interface. *NoFTL* is implemented in Shore-MT, on top of the OpenSSD hardware research platform and a real-time data-driven Flash emulator. We validate live the real-time Emulator, and are able to showcase different Flash layouts throughout the demonstration. Under *NoFTL* Flash management algorithms can benefit from the richer resources of the host system. We demonstrate stable and predictable performance and an improvement of up to 2.4x under TPC-C. The speedup results from a reduced garbage collection overhead (2x less erases and copybacks) due to better database integration of FTL functionality. Furthermore, the low erase count under NoFTL effectively doubles the lifetime of the Flash storage. In addition, we demonstrate the utilisation of native Flash parallelism under *NoFTL*. With its Flash-aware DBMS writer assignment strategy NoFTL achieves 1.5x higher transaction throughput due to reduced Flash chip contention.

Acknowledgements. This work was supported by the DFG projects "Flashy-DB" and "Topology of Technology".

6. REFERENCES

[1] Going beyond ssd: The fusionio software defined flash memory approach, 2013.
[2] http://diaswww.epfl.ch/shore-mt/, 2013.
[3] N. Agrawal and e. A. Prabhakaran. Design tradeoffs for ssd performance. In *Proc. ATC*, pages 57–70, 2008.
[4] P. Bonnet, L. Bouganim, I. Koltsidas, and S. D. Viglas. System co-design and data management for flash devices. In *Proc. VLDB 2011*, 2011.
[5] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proc. SIGMETRICS'09*, 2009.
[6] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song. A survey of flash translation layer. *J. Syst. Archit.*, 55(5,):332 – 343, 2009.
[7] A. Gupta, Y. Kim, and B. Urgaonkar. Dftl: A flash translation layer employing demand-based selective caching of page-level address mappings. In *Proc. ASPLOS XIV*, pages 229–240, 2009.
[8] S. Hardock, I. Petrov, R. Gottstein, and A. Buchmann. Noftl: Database systems on ftl-less flash storage. *Proc. VLDB Endow.*, 6(12), 2013.
[9] Y. Kim, B. Tauras, A. Gupta, and B. Urgaonkar. Flashsim: A simulator for nand flash-based solid-state drives. In *In Proc. SIMUL'09*, pages 125–131, 2009.
[10] I. Koltsidas and S. D. Viglas. Data management over flash memory. In *Proc.SIGMOD '11*, 2011.
[11] S.-P. Lim, S.-W. Lee, and B. Moon. Faster ftl for enterprise-class flash memory ssds. In *Proc. SNAPI'10.*
[12] D. Ma, J. Feng, and G. Li. Lazyftl: A page-level flash translation layer optimized for nand flash memory. In *Proc. SIGMOD '11*, pages 1–12, 2011.
[13] D. Ma, J. Feng, and G. Li. A survey of address translation technologies for flash memories. *ACM Comput. Surv.*, 46(3):1–39, 2014.
[14] Oracle. A quantitative comparison between raw devices and file systems for implementing oracle databases. white paper. 2004.
[15] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D. K. Panda. Beyond block i/o: Rethinking traditional storage primitives. In *Proc. HPCA*, 2011.
[16] C. Park, W. Cheon, J. Kang, K. Roh, W. Cho, and J.-S. Kim. A reconfigurable ftl architecture for nand flash-based applications. *TECS*, 7(4):38:1–38:23, 2008.

Gumbo: Guarded Fragment Queries over Big Data

[Demo paper]

Jonny Daenen
Hasselt University
Diepenbeek, Belgium
jonny.daenen@uhasselt.be

Frank Neven
Hasselt University
Diepenbeek, Belgium
frank.neven@uhasselt.be

Tony Tan
Hasselt University
Diepenbeek, Belgium
tony.tan@uhasselt.be

ABSTRACT

We present GUMBO, a system for the efficient evaluation of guarded fragment queries on top of Hadoop and Spark. A key asset of GUMBO is the reduced number of jobs in comparison with recent systems such as Pig, Hive or Shark. For unnested guarded fragment queries, GUMBO even provides a constant bound on the number of jobs independent of the size of the query. In the demo, we will address the following features of GUMBO: ease-of-use, query plan construction and visualisation, and query execution.

Categories and Subject Descriptors

[**Information systems**]: MapReduce-based systems, relational parallel and distributed DBMSs, key-value stores, relational database model

General Terms

DBMS

Keywords

MapReduce, Hadoop, Spark, Guarded-fragment Queries

1. INTRODUCTION

Recent years have seen a massive growth in parallel and distributed computations based on the use of the key-value paradigm. This proliferation was fostered by the emergence of popular systems such as Hadoop [14] and Spark [1]. Recent systems such as Hive [13], Pig [9], Shark [15], etc. provide an SQL-like query language on top of Hadoop and Spark. In this demo we showcase a novel system, called GUMBO,[1] that also operates on top of Hadoop and Spark, and is specifically tailored for the evaluation of so called *guarded fragment* (GF) queries. We show that, in general, GUMBO takes less jobs to evaluate GF queries than Pig, Hive or Shark.

[1]In case you are wondering about the name, GUMBO's brother is an elephant featuring in several animated movies but not known to be interested in guarded fragment queries.

2. GUARDED-FRAGMENT QUERIES (GF)

We briefly review the definition of guarded fragment (GF) queries. They are defined inductively as follows:

- Every atomic formula $S(\bar{x})$ is a GF query.
- If $R(\bar{x},\bar{y})$ is an atomic formula and $\psi_1(\bar{y},\bar{z}),\ldots,\psi_l(\bar{y},\bar{z})$ are GF queries, then the following $\varphi(\bar{x})$ is also a GF query:

$$\varphi(\bar{x}) \quad := \quad \exists\bar{y}\,\exists\bar{z}_1\,\cdots\,\exists\bar{z}_l \quad R(\bar{x},\bar{y}) \wedge \left(\begin{array}{c}\text{Boolean combination of}\\ \psi_1(\bar{y},\bar{z}_1),\ldots,\psi_l(\bar{y},\bar{z}_l)\end{array}\right)$$

The original definition of GF queries in [2] allows for 'ungarded negation' by including that if $\varphi(\bar{x})$ is a GF query, then so is $\neg\varphi(\bar{x})$. For pragmatic reason, the GUMBO system does not consider such ungarded negation. Take, for example, the negation of atomic relation $\neg R(\bar{x})$. Under the closed world assumption, its evaluation will involve collecting the active domain and performing a Cartesian product on them $|\bar{x}|-1$ times, which, in general, is a very expensive operation in distributed databases. We stress that GUMBO allows for guarded negation as is allowed in the definition above and is exemplified in the example in Section 5.

Originally GF queries were in [2] to investigate various properties of modal logic. Since then, they have become popular and found numerous applications in various fields. One example is the description logic $\mathcal{ALC}$, the basis of the knowledge representation in artificial intelligence as well as ontologies and web semantics. We refer the reader to [3, 11] and the references therein for more details. In fact, $\mathcal{ALC}$ itself is a subclass of GF queries [10]. Recent studies such as [6, 7] investigate the complexity of query answering in description logic. GF queries and its natural extension, *guarded negation* queries have also found applications in various database settings (for example, [4, 5, 12]).

To end this section, we sketch a scenario in which GF queries can be used. Consider a library that records which member borrows which books over a period of time. Specifically, there is a table R containing records with the following fields: `d`, `mem_id`, `b1`, `b2`, `b3`, `b4`, `b5`. Here `d` stands for date, `mem_id` for the member id, and each `b` represents a borrowed book.[2] Every night the new data that arose during the day

[2]In our imaginary library, each member can only borrow up

is added. To provide better service to its members, the librarian decided to find out more about the dynamics of the book transactions and starts exploring the data. For example, she may want to find out how (un)popular books about technology compared to other books. So she compiles a list S of the ISBNs of the books about technology, and uses the following query:

$$R(\texttt{d}, \texttt{mem\_id}, \texttt{b1}, \ldots, \texttt{b5}) \\ \wedge \neg S(\texttt{b1}) \wedge \neg S(\texttt{b2}) \wedge \neg S(\texttt{b3}) \wedge \neg S(\texttt{b4}) \wedge \neg S(\texttt{b5})$$

This query computes for each member all days that only non-technology books are borrowed..

3. GF QUERY EVALUATION IN GUMBO

We contrast GF query evaluation in Gumbo with that in Pig and Hive by means of an example.

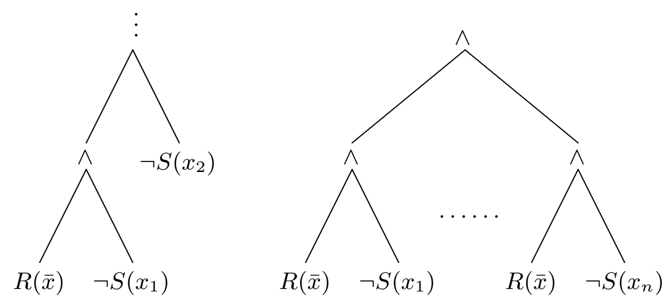
GF in Pig and Hive.

To implement a relational operation between two datasets, Pig requires us to perform a `cogroup`. Consider for instance the following query:

$$\varphi(\bar{x}) \ := \ R(\bar{x}) \wedge \Big(\neg S(x_1) \wedge \cdots \wedge \neg S(x_n)\Big),$$

where $\bar{x} = (x_1, \ldots, x_n)$. Here, R is an n-ary relation and S is a unary database relation.

Implementing the query φ in Pig/Hive/Shark will yield either one of the following execution plans:



The plan on the left requires n **rounds of shuffling** the datasets. Here, the first round evaluates $R(\bar{x}) \wedge \neg S(x_1)$, and the result is passed to the second round which in turn evaluates $(R(\bar{x}) \wedge \neg S(x_1)) \wedge \neg S(x_2)$, and so on. In contrast, the plan on the right requires **reading the input n number of times**: once for the evaluation of each $R(\bar{x}) \wedge \neg S(x_i)$.

In either plan, we are required to either read the same input multiple times or shuffling the datasets multiple times. This creates a bottleneck, since shuffling and reading the input can be very expensive, especially when the datasets are huge.

GF in Gumbo.

Gumbo operates on top of Hadoop as well as Spark and is specifically tailored to efficiently evaluate queries of the form:

$$\varphi(\bar{x}) \ := \ \exists \bar{y}\ \exists \bar{z}_1 \ \cdots\ \exists \bar{z}_l \\ R(\bar{x}, \bar{y}) \wedge \left(\begin{array}{c} \text{Boolean combination of} \\ S_1(\bar{y}, \bar{z}_1), \ldots, S_l(\bar{y}, \bar{z}_l) \end{array} \right) \quad (1)$$

where $S_1(\bar{y}, \bar{z}_1), \ldots, S_l(\bar{y}, \bar{z}_l)$ are atomic formulas. Gumbo can evaluate such queries in **two MapReduce jobs independent of l** and the form of the Boolean combination of $S_1(\bar{y}, \bar{z}_1), \ldots, S_l(\bar{y}, \bar{z}_l)$. In particular, Gumbo reads the input datasets only once. This should be contrasted with Pig and Hive where the number of jobs grows in the size of the number of Boolean combinations.

We call queries of the form (1) basic queries, i.e. when all $S_i(\bar{y}, \bar{z}_i)$ are atomic formulas. Gumbo can also evaluate multiple queries that depend on one another. For example, we can define $\varphi_1(x) = \exists y\ E(x, y) \wedge F(y)$, where $E(x, y)$ and $F(y)$ are atomic formulas, and $\varphi_2(z) = \exists y E(z, x) \wedge \varphi_1(x)$. In this case, φ_1 is a basic query, whereas φ_2 is a nested query, since it depends on the outcome of the query φ_1. In this case, Gumbo first takes two rounds to evaluate $\varphi_1(x)$, and then another two rounds to evaluate $\varphi_2(x)$. In general, to evaluate a set of queries in which the depth of dependency is m, Gumbo requires $2m$ dependent map-reduce jobs.

4. COMPONENTS OF GUMBO

Gumbo is written in Java and consists of the six components shown in Figure 1. We will briefly describe each of these components in the following paragraphs.

Parser. GF queries are provided together with the location of input and output relations. The queries are broken up into basic GF queries, i.e. queries of the form (1) and dependencies among them are determined. Structural errors such as cyclic dependencies are also checked here. The result of this component is a DAG, where the nodes are sets of basic GF queries to be evaluated, and the edges indicate the dependencies among the queries.

Partitioner. Given a DAG of basic GF queries as input, the partitioner aims to group queries in an optimal fashion in an effort to reduce the total number of parallel rounds. To this end, each query is assigned a *round number*, and all queries that have the same round number are grouped together. The result of this phase is a list of consecutive rounds each containing a set of basic GF queries.

The partitioner can greatly reduce the number of jobs as well as "balance" the computation load among the rounds. Gumbo approximates the computational load of a query by calculating or estimating the size of its input relations. We can show that, in general, obtaining the most optimal schedule is NP-hard, even if we assume that the computational load to evaluate each query is uniform. In our initial version of Gumbo, we therefore use a greedy algorithm to approximate the optimal schedule. This is a reasonable approach assuming that the queries are "few," or that the dependency depth is quite shallow. In the later versions of Gumbo, we plan to use an SMT solver (e.g. Microsoft's Z3 system [8]) to obtain the optimal schedule.

Job Constructor. Given a list of rounds, each round is compiled into two high-level map-reduce job as described below. Locations for intermediate files are also determined here.

to 5 books. If one member borrows less than 5 books, only the first few fields are assigned and the rest are assigned with the null value.

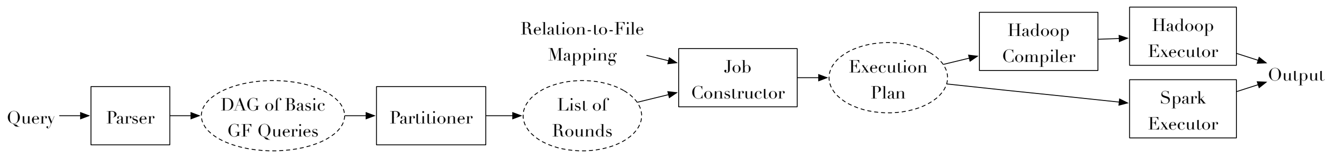


Figure 1: The GUMBO workflow.

The result of this phase is a GUMBO-plan.

For a query $\psi(\bar{x})$ of the form (1) GUMBO's job constructor builds the following mappers and reducers:

Mapper 1: On each tuple $R(\bar{a},\bar{b})$ Mapper 1 emits following the key-value pairs:

$$\langle S_1(\bar{c}_1) : R(\bar{a},\bar{b})\rangle, \ldots \langle S_l(\bar{c}_l) : R(\bar{a},\bar{b})\rangle$$

where each $\bar{c}_i$ is the projection of $(\bar{a},\bar{b})$ according to the $\bar{y}$ coordinates. The intuitive meaning of this part is that a tuple $R(\bar{a},\bar{b})$ *inquires* whether the tuple $S_i(\bar{c}_i,\bar{d})$ exists for some $\bar{d}$.

On each tuple $S_i(\bar{c},\bar{d})$ Mapper 1 emits the key-value pair $\langle S_i(\bar{c}) : \#\rangle$, where $\bar{c}$ is the projection of $(\bar{c},\bar{d})$ to the $\bar{y}$ coordinate The intuitive meaning of this part is that the key $S_i(\bar{c})$ *states* that the tuple $S_i(\bar{c},\bar{d})$ exists for some $\bar{d}$.

Reducer 1: For a key $S_i(\bar{c})$, the reducer operates on its set of values V as follows.

If # appears as value, for each value of the form $R(\bar{a},\bar{b})$, it emits a key-value pair $\langle R(\bar{a},\bar{b}) : i\rangle$, which means that $S_i(\bar{c},\bar{d})$ exists for some $\bar{d}$.

If # does not exist, for each value of the form $R(\bar{a},\bar{b})$, it emits $\langle R(\bar{a},\bar{b}) : -i)\rangle$, which means that there is no $\bar{d}$ such that $S_i(\bar{c},\bar{d})$ exists.

Mapper 2: This is an identity mapper that just reads and emits the key-value pairs created by Reducer 1.

Reducer 2: The keys are all of the form $R(\bar{a},\bar{b})$, with the associated values a subset of $\{-l,\ldots,-1,1,\ldots,l\}$. The values determine the Boolean assignment ξ, where ξ assigns $S_j(\bar{z}_j))$ with true, if and only if j appears among the values associated with $R(\bar{a})$. So, on key $R(\bar{a},\bar{b})$, the reducer evaluates the formula according to the assignment ξ. If it evaluates to true, it writes the tuple $\bar{a}$ into the output file.

In our implementation of GUMBO, we further optimize the algorithm above, such as by compressing the keys and values, thus, decreasing the number of data bytes to be shuffled.

Hadoop Compiler & Executor. This component takes a GUMBO-plan and compiles into a set of Hadoop map-reduce jobs using the mappers and reducers constructed above. The resulting plan can then be directly executed using Hadoop.

Spark Executor. This component takes a GUMBO-plan and executes the rounds one by one. The input data is stored in Spark's RDD data structure and the execution plan is translated into RDD's transformations and actions such as `flatMap()` and `groupByKey()` to execute the jobs described in the GUMBO-plan.

Assuming that the relations $R, S_1, \ldots, S_l$ are all dumped in a single RDD A, a straightforward translation of the algorithm above to one that uses Spark's RDD is as follows.

```
B = A.flatMapToPair(<Mapper 1>);
C = B.groupByKey();
D = C.flatMap(<Reducer 1>);
E = D.flatMapToPair(<Mapper 2>);
F = E.groupByKey();
G = F.flatMap(<Reducer 2>); // G is the output
```

To make the Spark's algorithm above more efficient, we incorporate a few optimization strategies. For example, Mapper 2 in the algorithm above basically does nothing, so it can be omitted.

5. DEMO OVERVIEW

The goal of the demo is to show how GUMBO can be used to evaluate GF queries on top of Hadoop/Spark and to give insight in how it compares to the existing systems Pig, Hive and Hadoop. This comparison can be done on several levels: the query design, the query plan (which gives insight in the workings of the system) and the query execution where performance really matters. In the demo, the users can do the following.

Input the queries and the dataset. Queries are written in standard logic notation, where `&`, `|`, `!` denote the `and`, `or` and `negation` operations, respectively. We use the standard symbol `=` to define the query. For example, the user can input the following queries, where $E(x,y)$, $F(y)$ and $G(x,z)$ are atomic formulas:

```
Out1(x)   = E(x,y) & !F(y) & G(x,z);
Out2(x)   = E(x,y) & Out1(y);
Out3(x)   = E(x,y) & Out1(y) & !Out2(x);
Out4(x,y) = E(x,y) & !Out1(x);

E(x,y)    - E.txt;
F(y)      - F.txt;
G(x,z)    - G.txt;

Out1(x)   - Out1.txt;
Out2(x)   - Out2.txt;
Out3(x)   - Out3.txt;
Out4(x,y) - Out4.txt;
```

The query `Out1(x)` collects all the `x`'s where for some `y,z`, the tuple `(x,y)` is in `E` and `(x,z)` is in `G`, but `y` is not in `F`. Similarly, the query `Out2(x)` collects all the `x`'s where for some `y`, the tuple `(x,y)` is in `E` and `y` is in `Out1`.

Users also indicate which relations should be read from disk and where these reside in the file system. In our example above, $E(x,y)$ is an atomic formula, so `E(x,y) - E.txt`

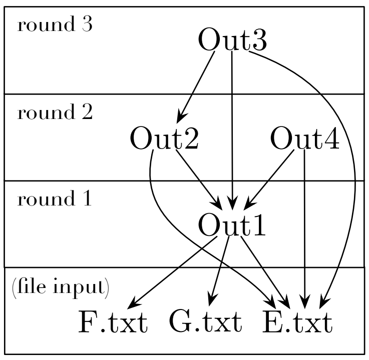


Figure 2: A DAG of jobs.

indicates that the relation E is to be read from the file `E.txt` To indicate where to write the output, we use the same format. For example, `Out1(x) - Out1.txt` indicates that the output of `Out1(x)` will be written in the file `Out1.txt`.

For the users to experiment, we provide a set of queries and for each query a collection of datasets on which the query behaves differently. For example, we provide some datasets in which the query outputs a lot of tuples, as well as some datasets in which the query outputs very few tuples.

Visualise the query plan. Similar to Pig, GUMBO also provides a "visualisation" of the dependencies among the jobs to evaluate the input queries. In our demo we are going to compare the map-reduce jobs constructed by GUMBO with those constructed by Pig.

In our example of `Out1`, `Out2`, `Out3` and `Out4` above, in GUMBO we obtain the DAG shown in Figure 2. GUMBO has two choices: evaluating the query `Out4` together with `Out2` or with `Out3`. If the computation load of `Out2` is much smaller than `Out3`, then the partitioner in GUMBO will merge `Out2` with `Out4`. In this case, the partitioner assigned the same round number to `Out2` with `Out4`, which means that they are to be evaluated simultaneously in the same map-reduce job.

Such visualisation provides the users the following benefits: (i) an insight in the plan construction; (ii) viewing the round numbers assigned by the partitioner; (iii) the effect of enabling/disabling certain optimizations (e.g. partitioners); (iv) comparison in the number of jobs in the query plans of GUMBO and those written in different systems such as Pig.

Execute the queries. In the final part of our demo, we will allow the users to execute some queries on sample data. We supply some sample-data of limited size, as "real" big data would require too much execution time.

The progress of a query and the log messages produced by the system can be viewed during execution. After the execution the user is able to inspect the output files to ensure that the queries were executed correctly and also the intermediate files to clarify the inner workings of the systems.

To further illustrate the inner working of GUMBO's Hadoop executor, we will show the content of the intermediate data passed from one round to the next, as well as some metrics such as the number of mappers and reducers used by GUMBO as well as by Pig and Hive. For the case of GUMBO's Spark executor, we will show the content of the RDD in the intermediate steps leading to the evaluation of the queries.

The key points that we want to highlight in this final part of the demo are: the time GUMBO takes to evaluate a query, and the performance gain obtained by combining multiple queries.

Acknowledgement

The third author is supported by FWO Pegasus Marie Curie fellowship.

6. REFERENCES

[1] Spark. http://spark.apache.org.
[2] H. Andréka, J. van Benthem, and I. Németi. Modal languages and bounded fragments of predicate logic. *Journal of Philosophical Logic*, 1998.
[3] F. Baader, D. Calvanese, D. McGuiness, D. Nardi, and P. Patel-Schneider, editors. *The Description Logic Handbook.* Cambridge University Press, 2003.
[4] V. Bárány, G. Gottlob, and M. Otto. Querying the guarded fragment. In *LICS*, 2010.
[5] V. Bárány, B. ten Cate, and M. Otto. Queries with guarded negation. *PVLDB*, 5(11):1328–1339, 2012.
[6] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Data complexity of query answering in description logics. In *KR*, 2006.
[7] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Tractable reasoning and efficient query answering in description logics: The *DL-Lite* family. *Journal of Automated Reasoning*, 39(3):385–429, 2007.
[8] L. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS*, pages 337–340, 2008.
[9] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanam, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a highlevel dataflow system on top of mapreduce: The pig experience. *PVLDB*, 2(2):1414–1425, 2009.
[10] E. Grädel. Description logics and guarded fragments of first order logic. In *DL*, 1998.
[11] I. Horrocks. Ontologies and the semantic web. *Commun. ACM*, 51(12):58–67, 2008.
[12] R. Rosati. On the decidability and finite controllability of query processing in databases with incomplete information. In *PODS*, 2006.
[13] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using hadoop. In *ICDE*, 2010.
[14] T. White. *Hadoop - The Definitive Guide: Storage and Analysis at Internet Scale (3. ed., revised and updated).* O'Reilly, 2012.
[15] R. Xin, J. Rosen, M. Zaharia, M. Franklin, S. Shenker, and I. Stoica. Shark: Sql and rich analytics at scale. In *SIGMOD*, 2013.

WAVEGUIDE: Evaluating SPARQL Property Path Queries

Nikolay Yakovets Parke Godfrey Jarek Gryz
York University
Toronto, Canada
{hush, godfrey, jarek}@cse.yorku.ca

ABSTRACT

The extension of SPARQL 1.1 of *property paths* now offers a type of *regular path query* for RDF graph databases. While eminently useful, these queries are difficult to optimize to evaluate efficiently. We have embarked on a project we call WAVEGUIDE to build a cost-based optimizer for SPARQL queries with property paths. WAVEGUIDE maps the property path to a *waveguide plan* (WGP) composed of *wavefront automata* (WFAs) modeled by (non-deterministic) finite automata. The waveguide plan *guides* the graph search during evaluation. Our WAVEGUIDE prototype illustrates the types of optimizations this approach affords and the performance gains that can be obtained.

1. INTRODUCTION

Graph data has quickly become prevalent with the rise of the Semantic Web, social networks, and data-driven exploration in life sciences. Natural and efficient ways are needed to query over the structure of the graph. *Regular path queries* (RPQs) offer a means to query for nodes connected via matching paths. Support for RPQs has been recently added in the SPARQL query language for RDF data in its latest version, 1.1, via *property paths.*[1]

While eminently useful, property-path queries are challenging to evaluate efficiently and to optimize well. We have embarked on a project that we call WAVEGUIDE to build a highly effective, full-fledged cost-based optimizer for SPARQL queries with property paths. Our approach uses guided search through the graph using *finite state automata* based upon the regular expression of the property path to guide. We are able to gain orders of magnitude performance improvement for many property-path queries, while maintaining comparable performance for others, as the leading SPARQL query engines.

Regular path queries have been considered ever since *semi-structured* data models were first introduced [1, 13]. The complexity of RPQs for graph databases particularly has been well studied [2, 3]. In [11], the idea of employing NFAs to guide search for RPQ evaluation is introduced. (The introduction of *product automata* in [13] can well be considered a precursor to this idea.) In [7], they investigate fixpoint evaluation for property paths. In [18], we considered a mapping of property paths to SQL queries with common table expressions (with SQL recursion). In [19], we present a precursor of WAVEGUIDE that explores fixpoint evaluation for property paths using SQL recursion.

WAVEGUIDE's strategy is based on an *iterative search algorithm* guided by a *query plan*, which we call a *waveguide plan* (WGP), composed of *wavefront automata* (WFAs) modeled by *non-deterministic finite state automata* (NFAs).[2] Within this framework, we are able to express complex query evaluation plans which involve multiple search *wavefronts* that iteratively explore the graph. The *states* (of the automata) of the WGP represent path queries in their own right. States of the plan are materialized selectively during evaluation which allows for re-use of intermediate results. We call such materialized states *path views.*

A SPARQL path query can be potentially evaluated by any number of WGPs. A good plan is the one that achieves a balance of *minimizing*

1. the *search space* that needs to be explored,
2. the *recomputation* of answers as much as possible (through re-use with path views), and
3. the degree of *caching* needed by the plan.

These objectives cannot be optimized independently of one another. To address this, we propose a cost-based method that selects the best WGP based on estimated total cost. Our ultimate goal is a cost-based optimizer for SPARQL queries for RDF databases of the same caliber as cost-based optimizers for SQL queries for relational databases.

The graph exploration for the query's evaluation is driven by an iterative search procedure that is effectively a fixpoint evaluation (semi-naïve and bottom-up [8, 12]). Three steps are performed each iteration: crank, reduce and union.

1. Crank expands the search wavefronts in the graph to produces a set of tuples (a *delta*).
2. Reduce eliminates the duplicates from a delta to counter unbounded computation on cyclic graphs.
3. Union selectively materializes delta into cache.

The iteration stops when no new tuples are produced (i.e., we reach the fixpoint).

Each search wavefront is guided by a *wavefront automa-*

[1]We consider SPARQL queries with *distinct*, so a pair of nodes is considered an answer if there *exists* a path between the pair in the graph that matches the regular expression.

[2]We name these *wavefronts* following the convention in [8].

ton (WFA), a finite state machine modeled on a non-deterministic finite automaton (NFA). Unlike NFAs, which are used as recognizers of regular expressions on strings, WFAs afford us a number of features related to evaluation of regular expressions on graphs such as use of *seeds*, *append/prepend* transitions, and *path views*.

We demonstrate our WAVEGUIDE prototype via a *query plan designer* for designing and viewing the plans and a *runtime visualizer and profiler* for tracing the guided search evaluation. The interactive demonstration over social-network and life-science datasets highlights the benefits—and the interesting challenges—with our methodology.

In §2, we posit a cost model, discuss costs that arise in property-path evaluation with respect to graph and query characteristics, and present optimization strategies. In §3, we overview the implementation of the WAVEGUIDE prototype. In §4, we present the demonstration scenario.

2. PLAN PERFORMANCE

For a given query, of course, there may be many ways to guide the search. Our *cost model*, in abstract, is essentially to predict the size of the *graph walk*—the number of triples from the RDF store (that is, labeled edges from the graph) that will be joined—during the search evaluation as guided by the plan. We summarize *search cost factors* that can affect the cost (properties of the graph and of resulting pre-paths computed during evaluation) and *optimization methods* that are enabled by waveguide plans which address the search factors, in turn.

2.1 Search Cost Factors

Properties of the graph and of the WGP chosen—thus, the guided search during evaluation in terms of the pre-paths that are computed—will determine the evaluation cost.

Search Cardinalities. The *wavefronts*, that we chose for the WGP determine during the search the intermediate results (pairs of nodes labeled by state, thus connected by valid pre-paths) that are collected each iteration. Just as with different join orders in relational query evaluation, different wavefronts will result in different intermediate delta sizes. These intermediate cardinalities can vary greatly over plans.

To reduce overall search size, we need to choose wavefronts that result in fewer edge walks. WGPs can be costed to estimate their search sizes based on statistics about the graph, such as 1-gram and 2-gram label frequencies. (Such graph statistics can be computed offline for this purpose.)

Solution Redundancy. Each node pair appears at most once in the answer, even if there are multiple paths between the node pair satisfying the query's regular expression. As such, answer-path redundancy arises from two sources. First, in dense graphs, solutions are re-discovered by following conforming, yet different paths. Second, nodes are revisited by following cycles in the graph. Thus, the same answer pair may be discovered repeatedly during evaluation. It is critical to detect such duplicate solutions early in order to keep the search size and search cache small.

Sub-path Redundancy. The paths justifying multiple answer pairs may *share* significant segments (*sub-paths*) in common. This arises, for instance, in dense graphs and with hierarchical structures (e.g., *isa* and *locatedIn* edge labels). Consider the query "?p :locatedIn+ Canada". Every person located in the Annex in Toronto qualifies, since the Annex is located in Toronto located in Ontario located in Canada. The sub-path "Annex :locatedIn+ Canada" is shared by the answer path for each Annex resident.

Because we keep only node-pairs (plus state) in the search deltas, and not explicitly the paths themselves,[3] we may walk these sub-paths many times, recomputing "Annex :locatedIn+ Canada" for each Annex resident.

2.2 Optimization Methods

We consider WGP optimization methods in relation to the search cost factors above.

Choice of Wavefronts. The direction in which we follow edges, and where we start in the graph, with respect to the regular expression will result in different *search cardinalities.* Our choice of WFAs in the plan dictates the wavefront(s).

Reduce. WAVEGUIDE's evaluation strategy is designed to counter solution redundancy. Redundancy of *candidate* solutions is addressed by removal of duplicates against both cache (cache) and delta (delta) by the reduce operation. As a further optimization, once a solution seed-target pair has been discovered, *first-path pruning* (fpp) removes the *seed* from further expansion by the search wavefronts.

Threading / Sub-queries. To counter *sub-path redundancy* requires us to decompose a query into sub-queries. We call this decomposition *threading*, and our waveguide plans accommodate this. The portion of the regular expression that will result in sub-paths that will be shared by many answer paths can be computed independently by a separate wavefront. Sub-path sharing can be predicted by graph statistics to indicate when sub-queries should be considered.

Partial Caching. Delta results are cached during evaluation as we need to check against the cache for redundantly computed pairs. For large intermediate cardinalities, this can be a significant cost. However, some of this cost can be negated. In particular, not every state in the plan's WFAs needs to have its node-pairs cached. Caching is only needed when redundancy is possible, due to cycles in a WFA or in the graph. States without cycles need not be cached.

2.3 An Illustration

We illustrate the impact of different plans (WGPs) on query evaluation over an example query

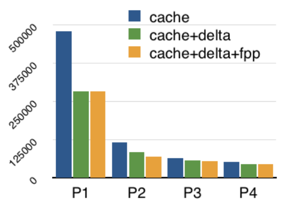
$Q =$?p :marriedTo/:diedIn/:locatedIn+/:dealsWith+ USA

over the real-world dataset YAGO2s [17] with 229M triples.

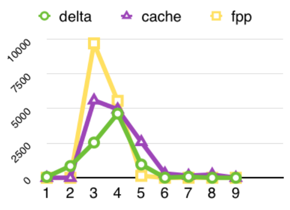
P_1: single prepending wavefront USA $\rightarrow$?p.
P_2: single appending wavefront ?p $\rightarrow$ USA.
P_3: two wavefronts and a join:
?p $\rightarrow$:locatedIn+ ?x
?x :dealsWith+ $\leftarrow$ USA.
P_4: P_2 but with a threaded sub-path
:locatedIn+/:dealsWith+ USA.

Fig. 1a shows the effects of wavefront choice on search cardinality. Note the order of magnitude difference between the best, P_4, versus the worst, P_1. The three types of redundancy pruning—cache, delta, and fpp—are illustrated for each plan. Fig. 1b plots search size across iterations for P_2 with pruning; over 40% of tuples are pruned. Fig. 1c plots delta sizes over iterations for P_1 and P_3. Note how the selective search of P_3 is better behaved than the rapid expansion of P_1. In Fig. 1d, the total execution time for each plan is

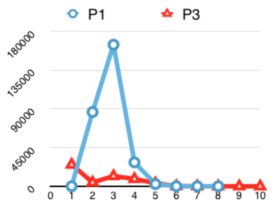
[3]Note this design choice in our evaluation strategy is critical for good performance, due to solution redundancy.



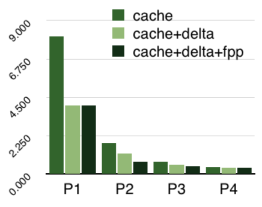
a) Search size for different plans and pruning types



b) Redundancy pruning (by type) over iterations of P2



c) Delta sizes over iterations



d) Total query time

Figure 1: Effect of plans on query evaluation.

presented.[4] This demonstrates the significant improvement in performance achievable by careful design of the WGP.

3. THE WAVEGUIDE PROTOTYPE

To demonstrate the efficacy of the WAVEGUIDE evaluation strategy, we focus on evaluation of SPARQL 1.1 property paths over large RDF graphs. We illustrate how WAVEGUIDE can be implemented effectively on a modern relational database system. We use POSTGRESQL due to that it is open-source and has a high-performance procedural SQL implementation. However, any RDBMS with good procedural SQL support could be used.

WAVEGUIDE's resource-intensive tasks can be delegated to POSTGRESQL via SQL and *procedural* SQL routines. This implementation of our methodology gains us high performance, scalability, and rapid deployment.

The architecture of our prototype is shown in Fig. 2. It consists of two layers: *application* and *RDBMS*. The application layer provides a user front-end, preprocessing the graph data, parsing user queries, generating WGPs, and visualizing key steps during the search. The RDBMS layer provides postprocessing of the graph data and performing the iterative WAVEGUIDE graph search for the given WGP.

We implement the application layer of the WAVEGUIDE prototype in Java. The layer consists of four main modules: a data importer, a query parser, a plan generator, and a data visualizer.

Data importer. This validates RDF data encoded in common formats (e.g., N-triples and Turtle.). It converts these to a tab-separated value format for bulk loading in the RDBMS.

Query parser. We use the Apache ANTLR open-source framework to parse SPARQL 1.1 property path query strings into an internal tree representation.

Plan generator. Given the query parse tree, we produce a

[4]The queries were run on a 2xXeon E5-2640v2 CPU server with 7200RPM HDD running Ubuntu Server 12.04 x64 and PostgreSQL 9.3.

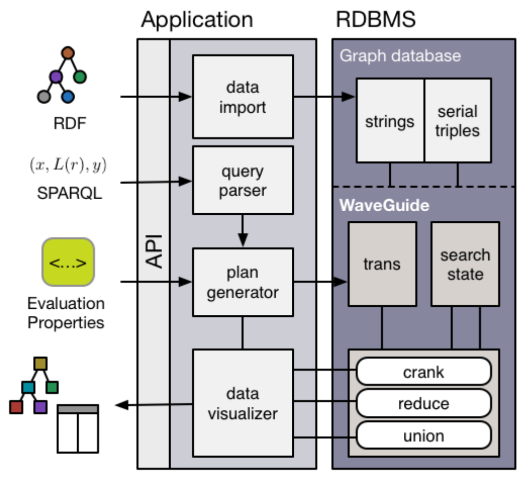


Figure 2: Overview of a prototype system

base WGP from an NFA that recognizes the regular expression of the query. We then employ a simple greedy WGP generation algorithm using the label cardinality estimates from the graph database. The produced WGP can be manually tuned by the end user via a *graphical evaluation plan designer* (shown on the left in Fig. 3).

Data visualizer. We employ the GRAPHSTREAM open-source library [5] to perform the graph visualization in our system. This allows us to visualize dynamically the key steps involved in the WAVEGUIDE search process. We interface with the RDBMS to visualize the search cache at each iteration of the crank, reduce, and union steps. To provide technical insight to the WAVEGUIDE process, we display a number of relevant evaluation parameters and statistics (shown on the right in Fig. 3).

The RDBMS acts both as the graph store and as the execution platform for WAVEGUIDE's iterative algorithms.

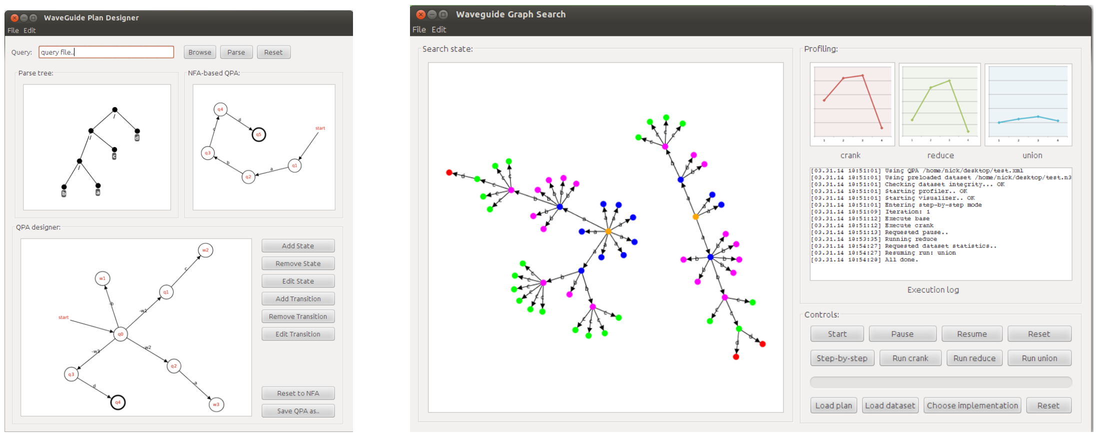
Graph database. We represent a graph database in a single logical triples table, which is decomposed into two physical tables—strings and a surrogate serialTriples—to reduce storage space and improve performance. The surrogate table is indexed in all six ways—spo, sop, pos, pso, osp, and ops—to accommodate the guided search.

Guided search. We implement the guided search process via a procedural SQL program. The WGP that guides the search process is encoded in the trans table. To improve the performance, the cache is stored in an unlogged, ephemeral searchCache table. This is indexed to cover the access paths used by the iterative search. We implement profiling functions here to feed evaluation statistics to the data visualizer.

4. DEMONSTRATION

We design scenarios for three demonstration *objectives* for demonstrating our prototype system: 1. *familiarization*, 2. *challenges*, and 3. *efficacy*. Due to the sizes of the datasets, we deploy the database layer of the prototype in the "cloud" in *Amazon EC*[2].

To *familiarize* researchers with our methodology, we demonstrate WAVEGUIDE evaluation of a number of simple property path queries over well-known RDF datasets such as FOAF [6] and DBPedia [4]. We construct the queries to



```
[03.31.14 10:51:01] Using QPA /home/nick/desktop/test.xml
[03.31.14 10:51:01] Using preloaded dataset /home/nick/desktop/test.n3
[03.31.14 10:51:01] Checking dataset integrity... OK
[03.31.14 10:51:01] Starting profiler.. OK
[03.31.14 10:51:01] Starting visualizer.. OK
[03.31.14 10:51:01] Entering step-by-step mode
[03.31.14 10:51:09] Iteration: 1
[03.31.14 10:51:12] Execute base
[03.31.14 10:51:12] Execute crank
[03.31.14 10:51:13] Requested pause..
[03.31.14 10:53:35] Running reduce
[03.31.14 10:54:27] Requested dataset statistics..
[03.31.14 10:54:27] Resuming run: union
[03.31.14 10:54:28] All done.
```

Figure 3: *Query plan designer* and *runtime visualizer and profiler.*

be fairly selective such that the whole evaluation process is comprehensible when visualized step-by-step.

To highlight the *challenges* of the proposed evaluation process, we design a number of non-trivial queries for various domains such as social networks (e.g., the LDBC Social Network Intelligence Benchmark [14]), life sciences (e.g., UNIPROT [16]), and encyclopedic (e.g., YAGO2s [17]). We present the audience with the query at hand, various statistics about the dataset, and show how to design an efficient evaluation plan using the capabilities offered by WAVEGUIDE's WGP mechanism. We focus on the interesting challenges for WGP optimization: efficient join order, cardinality estimation of simple and transitive paths, simplification of the guiding automaton, and intermediate data re-use.

To demonstrate the *efficacy* of WAVEGUIDE, we perform an online, interactive benchmark on a number of datasets and query loads against the native RDF-store Apache JENA [10]. We show that in many situations WAVEGUIDE outperforms JENA by several orders of magnitude.

5. REFERENCES

[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[2] P. Barcelo, L. Libkin, A. W. Lin, and P. T. Wood. Expressive languages for path queries over graph-structured data. *Transactions on Database Systems*, 37(4):31, 2012.

[3] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. Rewriting of regular expressions and regular path queries. In *Proceedings of the Symposium on Principles of Database Systems*, pages 194–204. ACM, 1999.

[4] The DBpedia knowledge base. http://dbpedia.org/.

[5] A. Dutot, F. Guinand, D. Olivier, Y. Pigné, et al. GraphStream: A tool for bridging the gap between complex systems and dynamic graphs. In *Emergent Properties in Natural and Artificial Complex Systems (Satellite Conference within ECCS)*, 2007.

[6] FOAF: The friend of a friend project. http://www.foaf-project.org/.

[7] A. Gubichev, S. J. Bedathur, and S. Seufert. Sparqling Kleene: Fast property paths in rdf-3x. In *Workshop on Graph Data Management Experiences and Systems*, pages 14–20. ACM, 2013.

[8] J. Han, G. Qadah, and C. Chaou. The processing and evaluation of transitive closure queries. In *Advances in Database TechnologyâĂŤEDBT'88*, pages 49–75. Springer, 1988.

[9] S. Harris and A. Seaborne. SPARQL 1.1 query language. W3C working draft. http://www.w3.org/TR/sparql11-query/, November 2012.

[10] Apache Jena. https://jena.apache.org/, 2013.

[11] A. Koschmieder and U. Leser. Regular path queries on large graphs. In *Scientific and Statistical Database Management*, pages 177–194. Springer Berlin Heidelberg, 2012.

[12] M. J. Mäher and R. Ramakrishnan. Déjà vu in fixpoints of logic programs. In *Proc. North American Conf. on Logic Programming*, pages 963–980, 1989.

[13] A. Mendelzon and P. Wood. Finding regular simple paths in graph databases. *SIAM Journal on Computing*, 24(6):1235–1258, 1995.

[14] M.-D. Pham, P. Boncz, and O. Erling. S3G2: A scalable structure-correlated social graph generator. In *Selected Topics in Performance Evaluation and Benchmarking*, pages 156–172. Springer, 2013.

[15] W3C: Resource Description Framework (RDF). http://www.w3.org/TR/rdf-concepts/, 2004.

[16] UniProt: Protein knowledgebase. http://www.uniprot.org/.

[17] YAGO2s: A high-quality knowledge base. http://yago-knowledge.org/resource/. Max Planck Institut Informatik.

[18] N. Yakovets, P. Godfrey, and J. Gryz. Evaluation of SPARQL property paths via recursive SQL. In L. Bravo and M. Lenzerini, editors, *AMW*, volume 1087 of *CEUR Workshop Proceedings*. CEUR-WS.org, May 2013.

[19] N. Yakovets, P. Godfrey, and J. Gryz. Waveguide: Toward cost-based evaluation of sparql property path queries. In *Proceedings of the 4th International Workshop on Semantic Search Over the Web VLDB*. ACM, 2014.

Meta-Stars: Dynamic, Schemaless, and Semantically-Rich Topic Hierarchies in Social BI

Enrico Gallinucci
DISI – Univ. of Bologna, Italy
enrico.gallinucci2@unibo.it

Matteo Golfarelli
DISI – Univ. of Bologna, Italy
matteo.golfarelli@unibo.it

Stefano Rizzi
DISI – Univ. of Bologna, Italy
stefano.rizzi@unibo.it

ABSTRACT

A key role in OLAP analyses of textual user-generated content for social business intelligence (SBI) is played by topics, i.e., concepts of interest within a subject area. Topic hierarchies are irregular, heterogeneous, dynamic, and possibly schemaless; besides, unlike in traditional OLAP, different semantics for topic aggregation can be envisioned. In this demonstration we present an architecture for SBI based on meta-stars, a novel approach to topic modeling in ROLAP systems. By coupling meta-modeling with navigation tables, meta-stars can cope with changes in the schema of irregular hierarchies and with schemaless ones; besides, they enable a new class of OLAP queries based on semantically-aware aggregation. The demonstration will focus both on the hierarchy update process and on the querying expressiveness.

1. INTRODUCTION

In the last few years, the success of social networks has led to the accumulation of a huge wealth of user-generated contents (UGCs) about people's tastes, thoughts, and actions —especially, those coming in the form of textual *clips*. This phenomenon is raising an increasing interest from decision makers because it can give them a fresh and timely perception of the market mood [1]. Unfortunately, though some commercial tools are available for analyzing textual clips using a few ad-hoc indicators, they do not support flexible and fully interactive analyses; besides, these tools are essentially built as self-standing applications and are not seen as a permanent part of the company information system.

To bridge this gap, *social business intelligence* (SBI) has emerged as the discipline of effectively and efficiently combining corporate data with UGC to let decision-makers analyze and improve their business based on the trends and moods perceived from the environment [2]. The goal of SBI is to enable powerful and flexible analyses for users with a limited expertise in databases and ICT; this is typically achieved by storing information into a data warehouse, in the form of multidimensional cubes to be accessed through OLAP techniques.

A key role in the analysis of textual clips is played by *topics*, meant as specific concepts of interest within the subject area [3]. Users are interested in knowing how many people talk about a topic, which words are related to it, if it has a good or bad reputation, etc. Thus, topics are obvious candidates to become a dimension of the cubes for SBI. Like for any other dimension, users are interested in grouping topics together in different ways to carry out more general and effective analyses —which requires the definition of a topic hierarchy that specifies inter-topic roll-up relationships so as to enable aggregations of topics at different levels. However, topic hierarchies are different from traditional hierarchies (like the temporal and the geographical one) in several ways. First of all, trendy topics are heterogeneous (e.g., they could include names of people, places, etc.) and change quickly over time (e.g., if at some time a group of politicians were discovered to be corrupt, a new Scandal class of topics would emerge during the following days), so a comprehensive schema for topics cannot be anticipated at design time and must be dynamically defined. For some topics a classification could even be hard due to their fuzzy nature, or unnecessary due to their transitoriness. Even when a schema is present, the expressiveness it requires is often beyond the one of the standard multidimensional model, i.e., topic hierarchies are non-onto, non-covering, and non-strict[1].

While these structural irregularities are already managed in some research models (e.g., [5]), handling hierarchies with dynamic schemata —or even potentially schemaless hierarchies— as required by topic hierarchies still constitutes a big challenge for SBI. Ontologies come to the rescue here, because the wide expressiveness they support enables an effective modeling of topic hierarchies with all their peculiarities; however, the problem of how to move this on a relational platform remains open.

To bridge the gap, in this demonstration we present a complete architecture for SBI based on *meta-stars* [2], a novel approach to topic modeling in relational OLAP systems. By coupling meta-modeled dimension tables with navigation tables, meta-stars can effectively cope with the peculiar requirements of topics hierarchies: on the one hand, meta-modeling enables hierarchy heterogeneity, schema dynamics, and schemaless hierarchies to be accommodated; on the other, navigation tables easily support non-onto, non-

[1]In a *non-onto*, *non-covering*, and *non-strict* hierarchy, instances can have different lengths, non-leaf topics can be related to facts, some hierarchy levels may be missing, and many-to-many relationship between topics may exist [5].

10.5441/002/edbt.2015.50

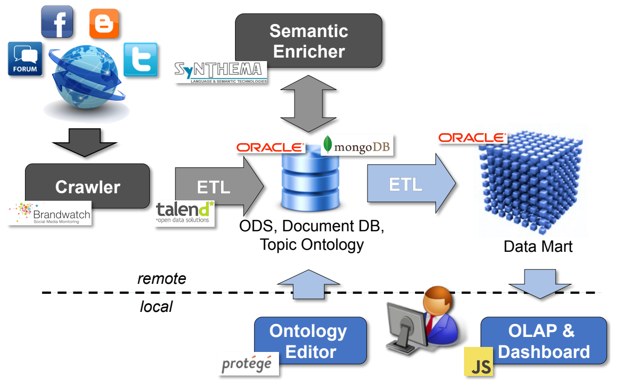


Figure 1: An architecture for SBI

covering, and non-strict hierarchies and also allow different roll-up semantics to be explicitly annotated, which in turn enables a brand new class of OLAP queries based on semantically-aware aggregation.

As already mentioned, topic hierarchies need be continuously updated and refined in both their schemata and instances to keep pace with the quickly-changing social environment. To enable simple and fast editing of hierarchies, we let users manage them in the form of ontologies. In this way, we can take advantage of existing ontology editors, which give good support to the design of irregular hierarchies. Besides, by creating a procedure that automatically loads/updates a meta-star starting from a given ontology, we relieve the user of the task of directly managing meta-stars and enable faster iterations of ontology design and testing.

2. SYSTEM OVERVIEW

The architecture used for this demonstration is depicted in Figure 1 and briefly commented in the following. The components in blue are those actively involved in the demo; the components running locally (on any Internet-enabled PC) and remotely are separated by a dashed line.

The user builds and refines the topic hierarchy by means of an *ontology editor*, then she launches an ETL process to automatically feed the meta-star within the social data mart (see Sections 2.1 and 2.2). The *crawler* component periodically runs a set of keyword-based queries over the web aimed at retrieving the clips (and the related meta-data) that are in the scope of the subject area. The textual content of the clips is then loaded into a document database, while the meta-data are loaded into an *operational data store* (ODS). The *semantic enricher* works on the document database to extract the semantic information hidden in the clip text and writes it in the ODS. A hybrid approach between supervised machine-learning [4] and lexicon-based techniques [6]) is adopted to extract the topics occurring within the single sentences of each clip, understand the syntactic relationships between words, and evaluate the sentiment related to each sentence and topic occurrence. An ETL process periodically extracts data about clips and topic occurrences and co-occurrences from the ODS and loads them into multidimensional cubes within the social data mart. Finally, the user uses OLAP tools and dashboards for flexibly analyzing clips and topics (Section 2.3). The total size of data involved in the demo (ODS + data mart) is about 1TB.

In our prototypical implementation of this architecture we use Brandwatch for keyword-based crawling, Talend for ETL, SyN Semantic Center by SyNTHEMA for semantic enrichment, Protégé for ontology editing, Oracle for storing the ODS, the topic ontology, and the social data mart, and MongoDB as the document database. For OLAP and dashboarding we developed an ad-hoc interface using JavaScript.

Of course, depending on the specific project context, lighter architectures could be sufficient (for instance, semantic enrichment may not be done if users are only interested in analyzing raw data). On the other hand, the architecture in Figure 1 can easily handle the data volumes normally involved in analyses, that in practice are often limited by either the diffusion of the subject area on the web or by the cost for buying clips from third parties.

2.1 Meta-Stars

Topics are first-class citizens for the large majority of relevant analyses that decision-makers find interesting in the field of SBI; thus, expressive and flexible solutions are required to model topics in multidimensional cubes. Meta-stars, introduced in [2], extend star schemata by enabling schemaless hierarchies to seamlessly coexist with hierarchies characterized by an irregular and dynamic schema, while supporting OLAP analyses. The basic idea is that it is almost impossible to devise a fixed schema for a subject area at design time and force all newly-discovered topics to fit that schema. However, a large part of topics can be effectively classified into levels, that mostly correspond to aggregation levels in traditional business hierarchies.

A *topic hierarchy* is an acyclic directed graph $H = (T, R)$, where T is a set of topics and R is a set of inter-topic roll-up relationships. A topic $t \in T$ can optionally be classified into a *level* $Lev(t)$, and a roll-up relationship $(t_1, t_2) \in R$ can be associated to a semantics $Sem((t_1, t_2)) \in \rho$ (with ρ being a list of user-defined roll-up semantics). The *meta-star* for a topic hierarchy H includes two tables:

1. A *topic table* storing one tuple for each topic in T. The schema of this table includes a primary surrogate key IdT, a Topic column storing the topic name, and a Level column storing the level, if any, in which the topic is classified.

2. A *roll-up table* storing one tuple for each arc in H^+. The tuple for arc (t_1, t_2) has two foreign keys, ChildId and FatherId, storing the surrogates of t_1 and t_2 respectively, and a column RollUpSignature that stores the *roll-up signature* of (t_1, t_2), i.e., a binary string of ρ bits where each bit corresponds to one roll-up semantics and is set to 1 if at least one roll-up relationship with that semantics is part of any directed path from t_1 to t_2, is set to 0 otherwise.

Remarkably, meta-stars defined as above directly support non-onto, non-covering, and non-strict hierarchies (because they pose no constraints on inter-level relationships), allow different roll-up semantics to be explicitly annotated (by storing roll-up signatures), and enable hierarchy heterogeneity and dynamics to be accommodated (by meta-modeling levels in the topic table).

Figure 2 shows an excerpt of the topic hierarchy we will use for the demonstration; the subject area is that of European political elections. Levels are represented by grey

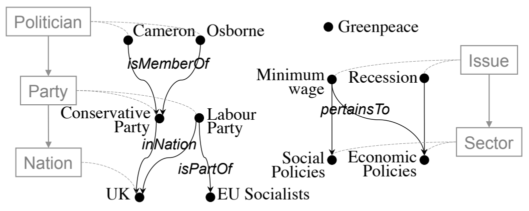


Figure 2: An excerpt of the topic hierarchy for European elections

TOPIC_T

IdT	Topic	Level
1	Cameron	Politician
2	Conservative Party	Party
3	EU Socialists	—
4	Minimum wage	Issue
5	Social Policies	Sector
6	Economic Policies	Sector
7	Greenpeace	—
8	UK	Nation
. . .	. . .	. . .

ROLLUP_T

ChildId	FatherId	RollUpSignature
1	1	0000
2	2	0000
. . .	. . .	0000
1	2	1000
1	8	1001
2	8	0001
4	5	0010
4	6	0010
. . .	. . .	. . .

Figure 3: Meta-star for European elections

boxes; topic Greenpeace is unclassified. In this example, the hierarchy is non-onto (also non-leaf topics such as UK can occur in clip sentences) and non-strict (the relationship between issues and sectors is many-to-many). Figure 3 shows a portion of the corresponding topic and roll-up tables where, for instance, the (transitive) relationship between Cameron and UK is expressed by the fourth tuple of the roll-up table, with roll-up signature 1001 (the list of roll-up semantics being $\rho =$ (isMemberOf, isPartOf, pertains, inNation)).

2.2 Feeding Meta-Stars

Clearly, managing topic hierarchies by directly editing topic and roll-up tables would be impractical. For this reason, in our approach topic hierarchies are modeled by users in the form of ontologies. Classes and instances are used to represent levels and topics, respectively, while properties are used to define roll-up relationships between topics. We provided a minimal framework for designing topic hierarchies by defining this set of superclasses and superproperties:

```
<Topic> <rdf:type> <owl:Class> .
<rollsUpTo> <rdfs:range> <Topic> .
<rollsUpTo> <rdf:type> <owl:ObjectProperty> .
<rollsUpTo> <rdfs:domain> <Topic>
```

In this framework, a level is defined as a class that specializes class *Topic* and a topic is defined as an instance of a level (unclassified topics are defined as instances of *Topic*). A roll-up relationship is first defined as a specialization of the *rollsUpTo* superproperty, and its domain and range are properly set considering the levels of its two end topics. Then, the roll-up relationship is implemented as an instance by linking the topics it involves.

The process of automatically generating a meta-star from an ontology takes advantage of the *Spatial and Graph* component in the Oracle DBMS, which allows to store and handle ontologies, as well as to integrate traditional SQL queries with SPARQL queries. Firstly, the ontology is exported from Protégé and loaded into the Oracle database. Then, a stored procedure is launched to read the ontology, determine the hierarchy schema (the levels and their relationships), and generate and execute the DML and the DDL SQL code that updates the data mart.

2.3 Querying Meta-Stars

While the aggregation semantics for OLAP queries is commonly understood and shared, in presence of irregular hierarchies —such as the topic one— some further possibilities arise. In particular, since facts (topic occurrences in clip sentences) can also be associated to non-leaf topics, multiple semantics of aggregation can be made available to users. To deal with these alternative semantics we extend the definition of OLAP query as follows.

Given topic hierarchy $H = (T, R)$, a *schema-aware topic query* is a triple of (i) a *group-by component*, that is a topic level l; (ii) an optional *selection*, that takes the form of a conjunction of Boolean predicates on topic levels; and (iii) a *semantic filter* σ consisting of a subset of allowed roll-up semantics, coded as a binary string of bits. The interpretation of a schema-aware topic query is that of building, for each topic t_i that has level l and satisfies the selection, a group of topics including all topics t such that the roll-up signature of (t, t_i) matches σ. Then, the facts for all topics included in each group are aggregated. Note that, while the group-by component and the selection determine which groups will be built, the semantic filter determines the composition of each group. Queries with $\sigma = 00\ldots0$ are called *queries without topic aggregation*, because the group for topic t_i only includes t_i; queries with $\sigma = 11\ldots1$ are called *queries with full topic aggregation* because all topics t from which t_i can be reached in H are included in the group for t_i. In all the other cases, we will talk of *queries with semantic topic aggregation* as topics are selectively aggregated based on the semantics of the roll-up relationships they are involved in.

Not all topics in T belong to a level, so there is a need for a further class of queries that work independently of the hierarchy schema. In a *schema-free topic query*, the topics of interest are explicitly listed in the group-by component, that takes the form of a set of topics $T' \subseteq T$. A group is built for each topic $t_i \in T'$, the composition of groups is determined like for schema-aware queries, so the same distinction based on topic aggregation can be made.

An example of a schema-aware query is the one asking for the number of occurrences of each Party topic for which Nation=UK, which can be done either without topic aggregation (only clips for the Conservative and Labour Parties are considered) or with topic aggregation (also clips for Cameron and Osborne are counted). An example of a schema-free query with semantic topic aggregation is the one asking for the number of occurrences of topic EU Socialists also considering the UGC mentioning parties of that group such as Labour Party but not the UGC mentioning politicians of those parties (filter on roll-up signature isPartOf).

3. DEMO SCENARIOS

As a case study to demonstrate the power of meta-stars we consider the incoming European elections. In particular, we focused the crawler on social networks, newspapers' websites, and politician's personal blogs from Italian, English, and German sources. In this context, three different scenarios will be demonstrated, yielding an overall demo duration of about 20 minutes.

The first scenario aims at showing how meta-stars can handle irregular and schemaless hierarchies. Non-strictness is shown by creating many-to-many relationships between issues and sectors, like in Figure 2 where Minimum wage per-

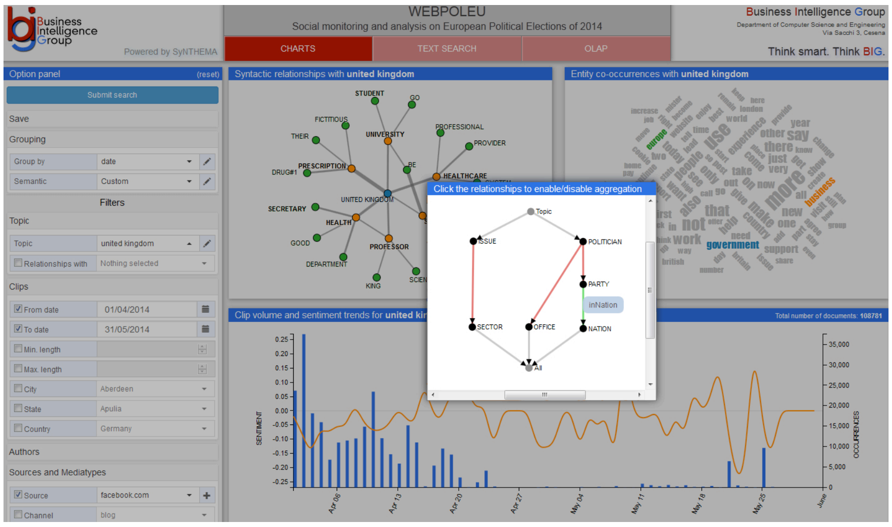


Figure 4: An excerpt of the topic hierarchy for European elections

tains to both Social Policies and Economic Policies. Meta-stars cope with non-strict hierarchies by simply having multiple tuples with the same ChildId in the roll-up table. To deal with non-coverage, a politician who supports the EPP without being member of any national party will be created; since the roll-up table contains every transitive relationship between topics, the problem of missing levels is simply overcame by directly coupling a child to its grandparent. Non-onto hierarchies are transparently accommodated because each topic (even non-leaf ones such as CDU) is represented in the topic table, so it can be directly referred from the fact table. Finally, a schemaless topic hierarchy is created by adding topics (such as Greenpeace in Figure 2) that do not belong to any level, i.e., have attribute Level set to null in the topic table. Note that also unclassified topics can be involved in roll-up relationships with other (classified or not) topics, and that these relationships can be transparently used for topic aggregation.

The second scenario is related to hierarchy dynamics. A recurrent situation in SBI is the discovery of new topics of interest and new topic levels, which requires to start a design iteration that refines the topic hierarchy and updates the meta-star accordingly. During the demo we will use the ontology editor to add new levels and topics, then launch the meta-star feeding procedure to let the new data be immediately available for querying. For instance, with reference to Figure 2, assume that the ontology initially does not include level Sector. Adding Sector and topics Social Policies and Economic Policies leads to update the meta-star as follows: (i) two new tuples are added to the topic table, with attribute Level set to Sector; (ii) the roll-up signature of each existing tuple in the roll-up table is extended with one bit to model the new pertainsTo semantic; and (iii) three tuples are added to the roll-up table to model the roll-up relationships between issues and sector. Note that, while the topic hierarchy has been modified intensionally, i.e., in its schema, the impact of this change on the meta-star level is purely extensional, i.e., it only involves the instances of the tables and not their schemata.

From the analyst point of view, meta-stars significantly increase the expressiveness of OLAP queries. The key element to this end is the roll-up signature, that allows topics to be aggregated by filtering the relationships the user wants to involve. So, the goal of the third scenario is to evaluate meta-stars from the point of view of querying effectiveness and efficiency. For instance, Figure 4 shows an analysis dashboard featuring the results of three different queries. In particular, the lower panel shows the volume and average sentiment for the daily occurrences of topic UK in Facebook clips written in April-May 2014. In the foreground window, the analyst is selecting a semantic filter on inNation to also include the clips mentioning the parties of UK; the SQL code generated for the final query is

```
SELECT     DT_DATE.date, AVG(FT.avgSentiment), COUNT(FT.occurrences)
FROM       TOPIC_T AS T, ROLLUP_T AS R, DT_DATE, DT_CLIP, FT
WHERE      FT.IdT = R.ChildId AND R.FatherId = T.IdT AND T.Topic = 'UK'
AND        BITAND(R.RollUpSignature,0001) = R.RollUpSignature
AND        <star join and selection predicates>
GROUP BY   DT_DATE.Date;
```

4. REFERENCES

[1] M. Castellanos and others. LCI: a social channel analysis platform for live customer intelligence. In *Proc. SIGMOD*, pages 1049–1058, Athens, Greece, 2011.

[2] E. Gallinucci, M. Golfarelli, and S. Rizzi. Meta-stars: multidimensional modeling for social business intelligence. In *Proc. DOLAP*, pages 11–18, S. Francisco, CA, 2013.

[3] L. García-Moya, S. Kudama, M. J. Aramburu, and R. B. Llavori. Storing and analysing voice of the market data in the corporate data warehouse. *Information Systems Frontiers*, 15(3):331–349, 2013.

[4] B. Pang, L. Lee, and S. Vaithyanathan. Thumbs up? sentiment classification using machine learning techniques. *CoRR*, cs.CL/0205070, 2002.

[5] T. B. Pedersen, C. S. Jensen, and C. E. Dyreson. A foundation for capturing and querying complex multidimensional data. *Inf. Syst.*, 26(5):383–423, 2001.

[6] M. Taboada, J. Brooke, M. Tofiloski, K. D. Voll, and M. Stede. Lexicon-based methods for sentiment analysis. *Computational Linguistics*, 37(2):267–307, 2011.

“I would like to watch something like ‘The Terminator’…” Cooperative Query Personalization Based on Perceptual Similarity

Christoph Lofi
Technische Universität Braunschweig
Mühlenpfordtstraße 23
D-38106 Braunschweig
lofi@ifis.cs.tu-bs.de

Christian Nieke
Technische Universität Braunschweig
Mühlenpfordtstraße 23
D-38106 Braunschweig
nieke@ifis.cs.tu-bs.de

ABSTRACT

In this paper, we showcase a privacy-preserving query personalization system for experience items like movies, music, games, or books. Personalizing queries for such items is notoriously difficult as meaningful query attributes are either missing in the database or would require extensive domain knowledge not available to most users. For this reason, state-of-the-art content provision platforms as e.g., Netflix or Amazon usually rely on recommender systems to support their users, and are often working in parallel with traditional SQL-style queries. Unfortunately, recommender systems have several shortcomings as for example high barriers for new users joining the system, which first have to setup a preference profile in a lengthy process, the inability to pose meaningful queries beyond recommendations matching the personal profile, and severe privacy concerns due to storing personal rating data for all users long-term. In order to provide an alternative, we present in this demonstration paper a powerful and intuitive query-by-example (QBE) interaction system. Bayesian Navigation is used to personalize a user’s query on the fly. The central challenge when using QBE is the selection of features to represent the items in the database. Here, we rely on a high-dimensional feature space which was mined from rating data of a large number of users, allowing us to measure perceived similarity between items to steer the query process. This also addresses many issues of recommender systems as our query capabilities can be used by any user anonymously in a drive-by fashion. In our proposed demo, users can try our never before presented system hands-on, and can use it to discover interesting movies tailored to their preferences with a pleasantly simple and enjoyable user experience.

Categories and Subject Descriptors

H.2.4 [Data Management]: Systems, Query Processing

Keywords

Personalization, Privacy, Perceptual Similarity, User Modeling, Query-By-Example, Bayesian Navigation.

1. INTRODUCTION

Effective personalization techniques have grown to be an integral and indispensable part of current information systems, and are essential to support users when faced with a flood of different choices. Here, two major approaches are common: a) Using SQL-style personalized queries on meta-data, which however require the users to have extensive domain knowledge in order to formulate precise and efficient queries. Additionally, SQL-style queries are difficult for the large domain of *experience items* like movies, books, music, or games, as the commonly available meta-data is often not describing the items in a suitable fashion (e.g. if they are funny, but with a dry humor, not slapstick). b) Adapting recommender systems which proactively suggest items to users based on their user profile, and which became particularly popular in systems like Amazon or Netflix [1], [2]: While many recommender systems provide recommendations of high quality [3], they have several shortcomings. Especially, for *each user* an elaborate user model needs to be built and stored, requiring up to hundreds of ratings until a user can effectively get meaningful recommendations. This creates a high barrier for new users to join the system. But more severely, this user model contains exhaustive personal information on a user’s preferences, her reaction to different items, or her general likes and dislikes. In order to query or use the system, this information must be *clearly associated* with the respective user and needs to be *stored long-term* for the system to work. Such profiles are highly valuable, and can be commercialized, abused, or even stolen. Obviously, this situation raises many privacy concerns and repels privacy-conscious users.

In this proposed demonstration, we therefore present an alternative approach fusing advantages of both recommender systems and SQL-based database personalization techniques, while at the same time avoiding many of the associated privacy risks. We realize this with privacy-preserving *query-by-example personalization*, which allows users to query for items fitting their current preferences easily without providing explicit feedback on attributes or their values. To achieve this, we utilize the *perceived similarity* between given items, which is harvested from user-item ratings, and transformed into a *perceptual space* [4]. However, we avoid the drawbacks of recommender systems: no user profiles are necessary to query the system, allowing situative, personalized, and anonymous ad-hoc queries.

2. SYSTEM DESIGN & FOUNDATIONS

In this section, we briefly outline the general design of our system, and provide some high-level insights into the theoretical foundations. This demonstration proposal is based on the work in [5]

10.5441/002/edbt.2015.51

where all theoretical foundations as well as the resulting performance evaluations are discussed in detail.

Basically, our approach allows users to easily explore a database with experience products by using personalized and privacy preserving query-by-example (QBE) navigation. In this proposed demonstrator, the database will contain around 12,000 movies. Users start the system interaction by providing an example of what they are currently looking for, e.g. "I enjoyed 'The Terminator', and I am now looking for a similar movie." Then, users are presented with a *display* of different items, and can provide a *feedback action* for some or all of them, resulting in a new display. This process continues until the user is satisfied with the results (a screenshot of our demo prototype is shown in Figure 1).

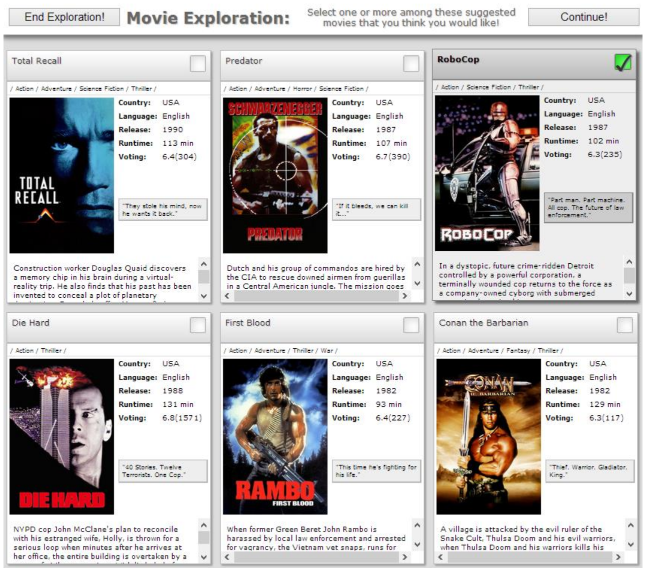
While this workflow could be achieved by simple similarity search, we strive to personalize this query process. Therefore, we build a disposable user profile which is only valid during the query's runtime, and the next display depends on the history of all feedback actions in the current query session instead of only the last feedback. Therefore, for two users, the same feedback action on one screen can result in different displays depending on their previous feedback within that query.

Accordingly, three major challenges are discussed in this section:

a) How can experience items as for example movies, books, music, games, software, or even hotels or restaurants be represented in a high-dimensional feature space such that meaningful similarity measurements and QBE navigation are possible?
b) How can we personalize an example-based query in such a way that it respects the user's feedback actions?
c) Which implications does such a system have on the user experience, and how does our approach compare to SQL-style systems and recommender systems?

Most popular QBE approaches in multi-media databases tried to operate on features generated from the actual multimedia file itself, which could either be low-features (e.g. color histograms or pattern-based features), or so-called high-level features as for example in scene composition [6] or content-based semantic features [7] (e.g., presence of explosions, or a mountain, or a flag, etc.) Here, our approach takes a completely different route, as our feature space results from external user ratings instead of being extracted from the media.

Information mined from user ratings has been shown to be very informative, and semantically more meaningful to users than traditional meta data as, e.g. information about the director or actors (as shown in e.g. [8] for movies). In our prototype, we demonstrate how such semantically rich rating data can represent each item of an experience product database within a high-dimensional feature space. The idea is that the resulting space implicitly encodes how users perceived a movie, e.g., if it was funny, or if certain plot elements or tropes were present. For this task, we adapt *perceptual spaces*. Perceptual spaces have been introduced in [4], and are built on the basic assumption that each user who provides ratings on items has certain personal interests, likes, and dislikes, which steer and influence her rating behavior [9]. For example, with respect to movies, a given user might have a bias towards furious action; therefore, she will see movies featuring good action in a slightly more positive light than the average user who doesn't care for action. The sum of all these likes and dislikes will lead to the user's overall perception of that movie, and will ultimately determine how much she enjoyed the movie, and how she rates it on a social movie site. Moreover, the rating will share this bias with other action movies in a systematic way. Therefore, one can claim that a perceptual space captures the "essence" of all user feedback, and represents


End Exploration!
Movie Exploration:
Select one or more among these suggested movies that you think you would like!
Continue!

Total Recall

/ Action / Adventure / Science Fiction / Thriller /

Country: USA
Language: English
Release: 1990
Runtime: 113 min
Voting: 6.4(304)

"They stole his mind, now he wants it back."

Construction worker Douglas Quaid discovers a memory chip in his brain during a virtual-reality trip. He also finds that his past has been invented to conceal a plot of planetary

Predator

/ Action / Adventure / Horror / Science Fiction /

Country: USA
Language: English
Release: 1987
Runtime: 107 min
Voting: 6.7(390)

"If it bleeds, we can kill it..."

Dutch and his group of commandos are hired by the CIA to rescue downed airmen from guerillas in a Central American jungle. The mission goes

RoboCop ✓

/ Action / Science Fiction / Thriller /

Country: USA
Language: English
Release: 1987
Runtime: 102 min
Voting: 6.3(235)

"Part man. Part machine. All cop. The future of law enforcement."

In a dystopic, future crime-ridden Detroit controlled by a powerful corporation, a terminally wounded cop returns to the force as a company-owned cyborg with submerged

Die Hard

/ Action / Thriller /

Country: USA
Language: English
Release: 1988
Runtime: 131 min
Voting: 6.8(1571)

"40 Stories. Twelve Terrorists. One Cop."

NYPD cop John McClane's plan to reconcile with his estranged wife, Holly, is thrown for a serious loop when minutes after he arrives at her office, the entire building is overtaken by a

First Blood

/ Action / Adventure / Thriller / War /

Country: USA
Language: English
Release: 1982
Runtime: 93 min
Voting: 6.4(227)

"This time he's fighting for his life."

When former Green Beret John Rambo is harassed by local law enforcement and arrested for vagrancy, the Vietnam vet snaps, runs for

Conan the Barbarian

/ Action / Adventure / Fantasy / Thriller /

Country: USA
Language: English
Release: 1982
Runtime: 129 min
Voting: 6.3(117)

"Thief. Warrior. Gladiator. King."

A village is attacked by the evil ruler of the Snake Cult, Thulsa Doom and his evil warriors, when Thulsa Doom and his warriors kills his

Figure 1. Screenshot of Prototype Implementation
First display, using "The Terminator (1984)" as start example

the shared as well as individual views of all users. A similar reasoning is also successfully used by recommender systems, e.g. [3], [10]. Now, the challenge of perceptual spaces is to reverse a user's rating process: For each item which was rated, commented, or discussed by a large number of users, we approximate the actual characteristics (i.e., the systematic bias) which led to each user's opinion as numeric features.

The general system design of our approach is shown in Figure 2: in an offline system initialization phase, a large number of user ratings (as e.g. obtained from a co-located recommender system, or from sites like e.g. IMDb or Netflix) is processed into a perceptual space. This process is described in detail in [4] and [5], but for clearer illustration of our demonstrator, we will briefly summarize it in the following. Then, personalized QBE queries are used to query that space.

Given is a large user-item-rating matrix, which usually is very sparse, containing only rating for around 1-2% of all user-item pairs. The goal is to find a matrix $A = (a_{m,k}) \in \mathbb{R}^{n_M \times d}$ representing movies as d-dimensional coordinates. To achieve this, we also need a helper matrix $B = (b_{u,k}) \in \mathbb{R}^{n_U \times d}$, representing user in the same space. Then, we use a factor model representing a rating function $f : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$. Basically, this function can predict missing ratings given user and item vectors. We approximate this function and the involved vectors/matrices, we use Euclidian Embedding (as in [11]), and we want the distance between a movie vector a_m and user vector b_u to be small if user u likes movie m; otherwise, it should be large. To account for general effects independent of personal preferences, for each movie m and user u, we introduce the model parameters δ_m and δ_u, which represent a generic movie rating bias relative to the average rating μ. Then, a rating of a movie m by a user u can be predicted by $\hat{r}_{m,u} = \mu + \delta_m + \delta_u - dis_E^2(a_m, b_u)$, i.e. it is average rating of all movies (e.g., μ=6.2 out of 1..10) plus the user bias (e.g., δ_u=-1.6 representing a critical user always rating worse than others) and the movie bias (e.g., it's a overall good movie with an average rating of 8.4, so δ_m=2.2). The last term, $dis_E(\cdot,\cdot)$, represents the distance of the movie vector and the user vector in a d-dimensional space. Finally, all movie vectors (and therefore the matrix A) can be approximated by solving a large least squares optimization problem with all instances of the above equation for which a rating is known (with some correcting terms accounting for noise added). Now, this matrix A represents our perceptual space.

Unfortunately, the resulting features in this space are implicit and have no direct real-world interpretation, therefore rendering SQL-style queries useless (i.e. you could ask for a value >0.8 on feature 5, but we don't know what feature 5 is). However, they allow for measuring perceived similarity effectively (i.e. the distance between the feature vectors). This now allows using the query-by-example paradigm, which provides simple query formulization without the need to explicitly refer to any features.

We adapt Bayesian Navigation [12] for this purpose. Here, for each query, a disposable user model is created during the interaction and discarded afterwards. Simplified, this model describes for each database item the probability that this particular item is the target item the user is looking for. Please note that usually the user is not explicitly aware of his target; if a user knew exactly which movie he was searching, an SQL-based query would be more efficient. However, we assume that there is at least one implicit target movie which the user simply does not know yet, nor can she describe exactly what her target looks like until she finds it. Our system is designed to guide a user to this implicit target, which is represented by adjusting the respective probabilities in each step. More formally, this is given by $P(M = M_i | H_t)$, i.e. the probability of a specific movie M_i being the intended implicit target M given the users current query history H_t. This history contains all displays (i.e. selection of movies a user has seen) and user actions on these displays (i.e. a binary selection of one or more of these movies). Together with a user prediction model, which predicts which movie a user will likely select out of a given display, (soft-min binary feedback in our case [12]), a selection strategy, which determines how the next display of the current interaction is selected, (most-probable strategy in our demo [5]), and a suitable start-up probability distribution, all probabilities for each movie can be recursively recomputed after each user interaction. This results in a new display and a new user interaction until the user is satisfied. Finally, a personalized list of top-k database items ranked by their probability is returned, and the temporary user model is disposed. The central challenge in computation is that Bayesian navigation requires an update of the modeled probabilities for each user interaction and each movie in the database, and each update of a single probability requires considering all other probabilities. This, of course, poses severe threats to the scalability of our system. Therefore, we presented an heuristic optimization technique in [5] which restricts these updates to the locality of the query. As a result, memory consumption and speed could be improved significantly.

The semantics of our approach are complemental to both SQL-style personalization as well as to recommender systems: recommender systems have precise knowledge of a user's likes and dislikes (basically: they "know" each user), and they use this knowledge to proactively provide personalized recommendations. However, they do not support dynamic queries, and cannot easily cater to changing moods and requirements. SQL-style personalization in contrast offers powerful queries on the "normal" available meta-data for users who know exactly what they are looking for. Our approach is in the middle-ground between both: for querying, a user just needs a vague idea / example of what she is looking for, and can navigate through items by simply pointing out good suggestions in the displays created using her feedback (which can be seen as situative recommendations). No direct interaction with attribute values is necessary. All three approaches serve their own purpose, and are useful in different situations.

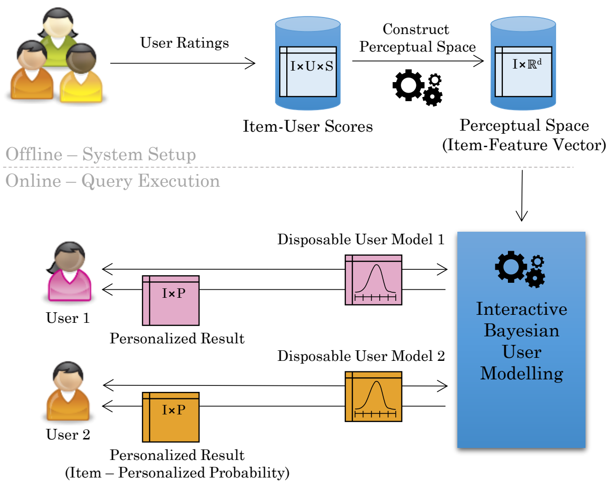


Figure 2. Basic System Design

3. PERSONALIZATION AND PRIVACY

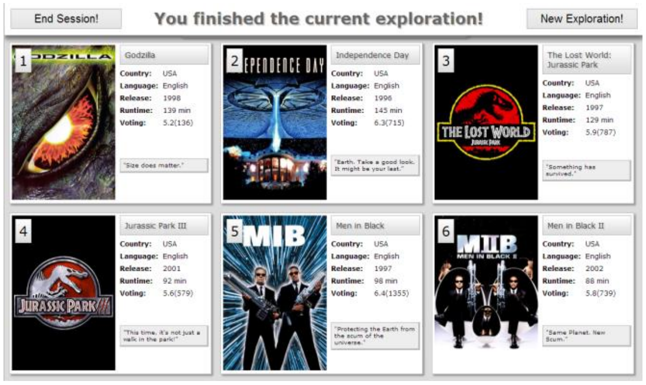
Privacy concerns severely impact a user's overall satisfaction with a Web-based system (as argued using the example of recommender system in [13]), and might even prevent them from using it altogether, if the balance between privacy concerns and perceived system utility becomes unfavorable.

The central focus of our system in terms of privacy is to allow all users to use the personalized query capabilities as anonymously as possible, without requiring a user profile or pre-query preference elicitation. Especially in contrast to recommender systems, this means that for browsing or querying, no long term user profiles are required – only feedback (selection history) with regard to the current query needs to be temporarily retained and does not have to be connected to a user id. This history of a query session is deleted after the query is completed, thus removing the need to store and protect this sensitive information. Even if a single query history was analyzed, you would need an extensive model of a user's preferences to convincingly match it to her, in which case you would not gain any new information from this query, and her remaining queries are not connected to this one.

But still, our system will require a small group of enthusiast users to provide identifiable rating data in order to construct the perceptual space, not unlike a recommender system. However, this construction process is completely decoupled from executing queries, and even the users which contributed ratings can later use the system anonymously for querying. The perceptual space itself does not contain any user related information, not even in an anonymized or masked form or even just the number of users that participated in its creation. It is basically just a matrix of movie ids and their major perceptual dimensions (n=100 in our case). Therefore, approaches de-anonymizing ratings similar to the ones detailed in [14] cannot be applied. This could allow a "trusted platform" like IMDb or MovieLens to use its users' ratings to construct a perceptual space, which then could be used by a 3rd party system like ours. In contrast to publishing anonymized rating data, publishing a perceptual space carries only minimal risks to the user's privacy. But in any case, even if users did decide to contribute ratings to build the space, all users can use the query capabilities of our system without leaving trails of personal information in an ad-hoc fashion.

4. EXPERIENCING THE DEMO SYSTEM

Our proposed demonstrator allows users to directly interact with the prototype implementation of our system. Users may freely issue



1 Godzilla
Country: USA
Language: English
Release: 1998
Runtime: 139 min
Voting: 5.2(136)
"Size does matter."

2 Independence Day
Country: USA
Language: English
Release: 1996
Runtime: 145 min
Voting: 6.3(715)
"Earth. Take a good look. It might be your last."

3 The Lost World: Jurassic Park
Country: USA
Language: English
Release: 1997
Runtime: 129 min
Voting: 5.9(787)
"Something has survived."

4 Jurassic Park III
Country: USA
Language: English
Release: 2001
Runtime: 92 min
Voting: 5.6(579)
"This time, it's not just a walk in the park!"

5 Men in Black
Country: USA
Language: English
Release: 1997
Runtime: 98 min
Voting: 6.4(1355)
"Protecting the Earth from the scum of the universe."

6 Men in Black II
Country: USA
Language: English
Release: 2002
Runtime: 88 min
Voting: 5.8(739)
"Same Planet. New Scum."

Figure 3. Screenshot of Prototype Implementation
Best matches movies after query is completed

their own queries, and may explore the system as they wish, using a pleasantly simple query interface. The prototype will include a wide selection of 11.976 movies from early 1900s up to 2005, and the underlying perceptual space was created by analyzing more than 103M user-movie ratings from over 480k users.

The users can interact with the system using a web-based user interface, which will resemble the screenshot in Figure 1. Additionally, we will prepare some query scenarios which highlight some interesting semantic aspects of our system. We will provide the required hardware and software to allow users to test the system, including at least two client devices. We also invite users to use their own mobile devices to access our web service. According to the user study we performed in [5], using our system was considered being an enjoyable and fun experience by most of the ~180 participating users.

Users can start a query by either providing a free example themselves, or by using one of twelve hand-picked examples provided by the system. In each display, 9 movies are displayed together with their respective poster, short synopsis, and general Meta data. From this display, users can select any number of movies as a positive example for the general "direction" in which they want to continue the query process. Then, after users are satisfied with the movies they encountered during the interaction, the query can be closed, and the final query result is presented in form of a list of movies ordered by their final Bayesian probability (see Figure 3).

Of course, we will also provide posters which explain the system design in detail, and discuss our design decision with interested visitors. This covers the general architecture, and also the theory behind building perceptual spaces, the Bayesian navigation technique, and the applied optimization techniques.

5. SUMMARY AND OUTLOOK

In this demonstration proposal, we described a personalized and privacy preserving query-by-example system for experience products as for example movies, books, or music. Our innovative system uses perceptual spaces built from a large number of user-item ratings to represent all database objects in a high dimensional feature space. We used Bayesian Navigation to allow users to issue queries in this space. The resulting system is thus very well suited to support users who only have a vague idea about what they are looking for, and helps them to explore the space in a personalized and guided fashion. Therefore, our system perfectly complements the features of SQL-based systems as well as those provided by recommender systems (which either support the case that the user knows exactly what she is looking for, or the case she does not know at all and therefore relies on a proactive recommendation).

In our prototype implementation, users can freely issue queries to a database containing around 12,000 movies in order to discover new and interesting titles, while at the same time learning about the inner working of our system.

6. REFERENCES

[1] G. Linden, B. Smith, and J. York, "Amazon.com recommendations: item-to-item collaborative filtering," *IEEE Internet Comput.*, vol. 7, no. 1, pp. 76–80, Jan. 2003.

[2] R. M. Bell, Y. Koren, and C. Volinsky, "All together now: A perspective on the Netflix Price," *CHANCE*, vol. 23, no. 1, pp. 24–24, Apr. 2010.

[3] Y. Koren and R. Bell, "Advances in Collaborative Filtering," in *Recommender Systems Handbook*, 2011, pp. 145–186.

[4] J. Selke, C. Lofi, and W.-T. Balke, "Pushing the Boundaries of Crowd-Enabled Databases with Query-Driven Schema Expansion," *Proc. VLDB*, vol. 5, no. 2, pp. 538–549, 2012.

[5] C. Lofi and C. Nieke, "Exploiting Perceptual Similarity: Privacy-Preserving Cooperative Query Personalization," in *Int. Conf. on Web Information System Engineering (WISE)*, 2014.

[6] H. Sundaram and S.-F. Chang, "Computable scenes and structures in films," *IEEE Trans. Multimed.*, vol. 4, no. 4, pp. 482–491, 2002.

[7] S.-Y. Neo, J. Zhao, M.-Y. Kan, and T.-S. Chua, "Video Retrieval Using High Level Features: Exploiting Query Matching and Confidence-Based Weighting," *Lect. Notes Comput. Sci.*, vol. 4071, pp. 143–152, 2006.

[8] I. Pilászy and D. Tikk, "Recommending new movies: even a few ratings are more valuable than metadata," in *ACM Conf. on Recom. Systems (RecSys)*, 2009.

[9] D. Kahneman and A. Tversky, "Psychology of Preferences," *Sci. Am.*, vol. 246, no. 1, pp. 160–173, 1982.

[10] T. Hofmann, "Latent semantic models for collaborative filtering," *ACM Trans. Inf. Syst.*, vol. 22, no. 1, pp. 89–115, Jan. 2004.

[11] M. Khoshneshin and W. Street, "Collaborative filtering via euclidean embedding," in *4th ACM Conf. on Recommender Systems (RecSys)*, 2010.

[12] I.J. Cox, M. L. Miller, T. P. Minka, T. V. Papathomas, and P. N. Yianilos, "The Bayesian Image Retrieval System, PicHunter: Theory, Implementation, and Psychophysical Experiments," *IEEE Trans. Image Process.*, 2000.

[13] B. P. Knijnenburg, M. C. Willemsen, Z. Gantner, H. Soncu, and C. Newell, "Explaining the user experience of recommender systems," *UMUAI*, vol. 22, no. 4–5, pp. 441–504, 2012.

[14] A. Narayanan and V. Shmatikov, "Robust De-anonymization of Large Sparse Dataset," in *IEEE Symposium on Security and Privacy*, 2008.

Liquid Benchmarking: A Platform for Democratizing the Performance Evaluation Process

Sherif Sakr [†,‡], Amin Shafaat [†], Fuad Bajaber [*]
Ahmed Barnawi [*], Omar Batarfi [*], Abdulrahman Altalhi [*]
[†]University of New South Wales, Sydney, Australia
{ssakr,ashafaat}@cse.unsw.edu.au
[*]King Abdulaziz University, Jeddah, Saudi Arabia
{fbajaber,ambarnawi,obatarfi,ahaltalhi}@kau.edu.sa
[‡]King Saud bin Abdulaziz University for Health Sciences, National Guard, Riyadh, Saudi Arabia

ABSTRACT

Performances evaluation, reproducibility and benchmarking represent crucial aspects for assessing the practical impact of research results in the computer science field. In spite of all the benefits (e.g., increasing impact, increasing visibility, improving the research quality) that can be gained from performing extensive experimental evaluation or providing reproducible software artifacts and detailed description of experimental setup, the required effort for achieving these goals remains prohibitive. In practice, conducting an independent, consistent and comprehensive performance evaluation and benchmarking is a very time and resource consuming process. As a result, the quality of published experimental results is usually limited and constrained by several factors such as: limited human power, limited time, or shortage of computing resources.

We demonstrate *Liquid Benchmarking* as an online and cloud-based platform for democratizing the performance evaluation and benchmarking processes. In particular, the platform facilitates the process of sharing the experimental artifacts (software implementations, datasets, computing resources, benchmarking tasks) as services where the end user can easily create, mashup, run the experiments and visualize the experimental results with zero installation or configuration efforts. In addition, the collaborative features of the platform enables the user to share and comment on the results of the conducted experiments so that it can guarantee a transparent scientific crediting process. Furthermore, we demonstrate four benchmarking case studies that have been implemented using the Liquid Benchmarking platform on the following domains: XML compression techniques, graph indexing and querying techniques and string similarity join algorithms.

1. INTRODUCTION

The last two decades have seen significant growth in the number of scientific research publications. One of the distinguishing characteristic of computer science research is that it produces *artifacts* in addition to the scientific publications, in particular *software implementations*. In general, reproducibility of experimental results represents a cornerstone in the computer science research field [2]. In practice, several benefits can be gained from providing reproducible experimental results including the improvement of the research quality, the gain of scientific credibility in addition to increasing the research visibility and the impact [2]. In particular, in an ideal world of computer science research, researchers describe the core of their contributions in the paper and then publicly provide the experimental datasets and the source codes/binaries of their software implementation for the community in order to facilitate the reproducibility of the published results in their publication. However, the world is not always ideal. While most of the computer science research literature usually present experimental results that evaluate/compare their proposed scientific contributions, the quality of such experimental results are usually limited due to several factors including: insufficient effort or time, unavailability of suitable test cases or any other resource constraints [8]. Furthermore, researchers used to focus on reporting about the *sweet spots* of their work in a way that is usually do not cover the ultimate picture or the practical insights of the real-world scenarios or the different application domains.

In principle, conducting an *independent* and *comprehensive* benchmarking study for the-state-of-the-art in a certain research topic is usually a very useful but a very challenging task as well. In particular, it usually consumes a lot of time and efforts due to multiple factors such as: unavailability of standard benchmarking tasks, lack of access to the implementations (source code or binaries) for some techniques which are proposed in the research literature in addition to the constraints of getting an access to different configuration of computing resources/environments that reflect the wide spectrum of different real-world scenarios [8]. Therefore, it is, unfortunately, quite common in several research domains to have no or little objective knowledge regarding the pros and cons of any set of different proposed research approaches/techniques which are sharing the goal of tackling a specific research challenge.

10.5441/002/edbt.2015.52

Recently, the challenge of defining and conducting comprehensive performance evaluations and benchmarking studies has been recognized by different research communities. In addition, several conferences, publishers and funding agencies have started to encourage their authors to provide the descriptions and the software artifacts that can facilitate the reproducibility of the experimental results of their publications. For example, in the database research community, ACM SIGMOD 2008 was the first conference that offered to verify the repeatability of the published experiments by allowing the authors to submit their programs and experimental datasets [6]. In addition, since 2008, the VLDB conference has created a new experimental and analysis track that encourages the research community to publish manuscripts that report and document thorough experimental evaluation and benchmarking studies[1]. Furthermore, several proposals [1] and tutorials have been presented in the major database venues to promote the crucial importance of performance evaluation, reproducibility and benchmarking in database research [2, 5]. Other research communities have been following the same approach such as the Semantic Web[2,3,4], Semantic Web Service[5], Business Process[6], Information Retrieval[7] in addition to the general Executable Paper Grand Challenge[8]. Although such types of research publications and benchmarking efforts are useful and important, however, they suffer from a main limitation which is that they present particular *snapshots* for the state-of-the-art that reflect the status at the time of their execution. In practice, the state-of-the-art in any research domain is always *dynamic* and *evolving* by default. For instance, new techniques that address the same research challenge of a previously published snapshot paper can be introduced or the performance characteristics of previously evaluated techniques can differ. Thus, such papers can be outdated shortly after they have been published.

In this paper, we demonstrate *Liquid Benchmarking* [8] as an online, collaborative and cloud-based platform that seeks to remedy the above mentioned challenges and problems by facilitating the *democratization* and improving the quality of the performance evaluation and benchmarking processes in the computer science research domain. In particular, we summarize the main strengths of our platform as follows:

- The platform dramatically reduces the time and effort for conducting performance evaluation process by facilitating the process of sharing the experimental artifacts (software implementations, datasets, computing resources, benchmarking tasks) and enabling the users to easily create, mashup and run the experiments with zero installation or configuration efforts.
- The platform supports for searching, comparing, analyzing and visualizing (using different built-in visualization tools) the results of previous experiments.
- The users of the platform can subscribe to get notifications about the results of any new running experiments for the domains/benchmarks of their own interests.
- The social and collaborative features of the platform enables turning the performance evaluation and benchmarking process into a *living* process where different users can run different experiments, share the results of their experiments with other users in addition to commenting on the results of the conducted experiments by themselves or by other users of the platform. Such features guarantee the utilization of the *wisdom of the crowd*, the *freshness* of the results, the establishment of a *transparent* process for scientific *crediting* and the development of scientific advances that trust and build on previous research contributions.

2. PLATFORM DESIGN

2.1 Underlying Technologies

The features and design decisions of the Liquid Benchmarking platform combine the facilities provided by different technologies as follows:

- *Software-as-a-Service*: The platform relies on the RESTful architectural style as an effective software distribution mechanism in which software implementations get hosted on the computing environments and made available as web services to the end-users over the Internet. Such mechanism requires zero downloading, installation or configuration effort at the side of the end user where all communication with software can be achieved using HTTP methods.
- *Cloud Computing*: The platform utilizes cloud computing as an effective technology for broad sharing of hardware resources and computing environments via the Internet. In particular, virtualization is a key technology of the cloud computing paradigm that improves the manageability of hardware resources by flexibly allowing computing resources to be provisioned on demand (in the form of virtual machines) and hiding the complexity of resource sharing details from cloud users. In practice, conducting a fair and *apples-to-apples* comparison between any competing software implementations requires performing their experiments using *exactly* the same computing environment [8]. In addition, performing a comprehensive and insightful evaluation process that assess different performance characteristics of the evaluated software implementations may require using several virtual machines with variant and scaling (in terms of computing resources) configuration settings (e.g. main memory, disk storage, CPU speed) that reflect different real-world scenarios [8]. The Liquid Benchmarking platform utilizes the virtualization technology for maintaining the testing computing environments in cloud platforms in the form of pre-configured *virtual machines* (with different configurations) which are hosting the competing software implementations (in the form of web services) and are shared by the end-users of the benchmark.
- *Collaborative and Social Software*: The platform is enabled with different Web 2.0 capabilities (e.g. user comments, tagging, forums) that support human inter-

[1]http://www.vldb.org/pvldb/vol1.html
[2]http://challenge.semanticweb.org/
[3]http://2014.eswc-conferences.org/important-dates/call-challenges
[4]http://iswc2014.semanticweb.org/call-replication-benchmark-data-software-papers
[5]http://sws-challenge.org/wiki/index.php/Main\textunderscorePage
[6]http://processcollections.org/past/2013-2/matching-contest
[7]http://www2.informatik.hu-berlin.de/~wandelt/searchjoincompetition2013/
[8]http://www.executablepapers.com/

action and facilitates the building of online communities between groups of researchers who share the same interests (peers) where they can interact and work together in an effective and productive way. Most important, the platform supports sharing the performance evaluation and benchmarking artifacts (e.g., software implementations, datasets, virtual machines) in a *workable* environment.

2.2 Benchmark Specifications

In Liquid Benchmarking, each benchmark is configured by defining the following main components:

- *Evaluated Solutions*: Represent the set of competing software implementations (e.g. algorithms, techniques, systems) which are sharing the goal of tackling the subject research challenge of the benchmark. The implementation of each evaluated solution needs to be wrapped with a web service interface before being integrated on the benchmark.
- *Service Schema*: Defines the set of parameters (inputs and outputs) that need to be defined for interfacing with the services of the evaluated solutions.
- *Task(s)*: Describes an operation which is specified for evaluating the competing implementations (e.g. queries, update operations, compression operations). In particular, each task represent an *instantiation* for the parameters of the service schema with a set of value that describes the specification of the task.
- *Metric(s)*: Represents a measure (e.g. execution time, response time, throughput) for evaluating the competing software implementations in performing the benchmarking tasks. In particular, it provides the basis for comparing the competing software implementations.
- *Testing Environment(s)*: Represents a set of resources configuration (e.g., CPU, disk, memory) for a computing environment (virtual machine) that hosts the services of competing software implementations.

2.3 Platform Components and Architecture

Figure 1 illustrates the architecture of the Liquid Benchmarks platform which are equipped with several *components* that are described as follows:

- **Web-based User Interface**: This component provides the end user with a user-friendly interface where he/she can *mash up* the components (e.g., services, tasks, metrics, computing environments) of the experiment in a *drag and drop* style. It also provides the end-user with other features such as: managing user account, maintaining the metadata store, searching and commenting on the results of previous experiments, subscribing to the results of a benchmark in addition to analyzing and visualizing the experimental results.
- **Metadata Store**: This component stores the information about the components (e.g., services, service schema, tasks, virtual machines) of the benchmark.
- **Experiment Manager**: The experiment manager receives the specification of the user-defined experiment, configured by the Liquid Benchmark UI, which is then registered for execution on the **Experiment Queue**. In principle, the experiment queue is used by the **Experiment Execution Engine** to ensure that the execution of one experiment in a testing environment is not going to influence the execution of another experiment in the same environment (an experiment can only start after the end of the current experiment, if exist, on the computing environment). Through the experiment life cycle, the **Experiment Execution Engine** sends a set of *notification events* to the **Notification Center** with the status of the experiment till its completion and storing its results in the **Repository of Experimental Results** for further analysis and visualization purposes. It should be noted that the **Experiment Execution Engine** is the component that is responsible for managing the life cycle of testing environments. In particular, it starts the virtual machine of a testing environment for running an experiment if it has been in a stopped mode or it stops the virtual machine if it has been idle for a while and has no pending experiments in the queue.

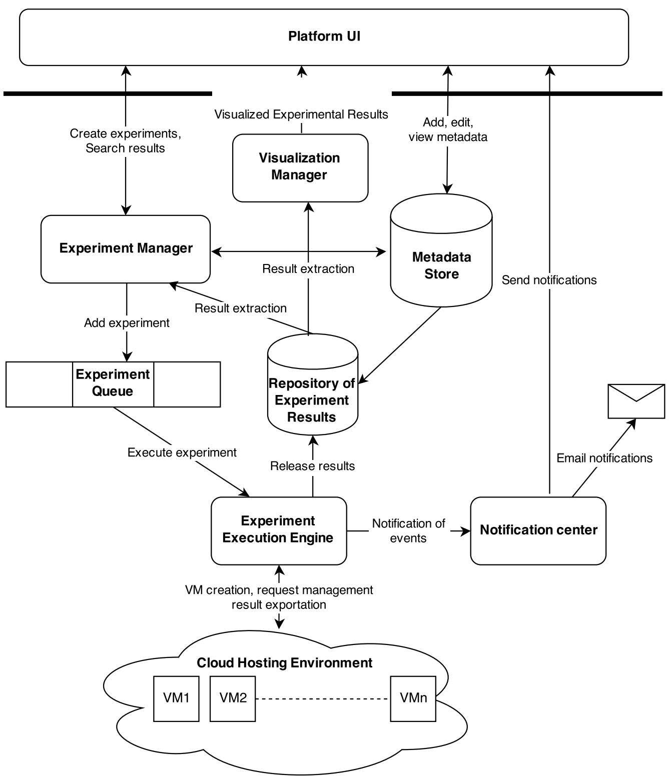
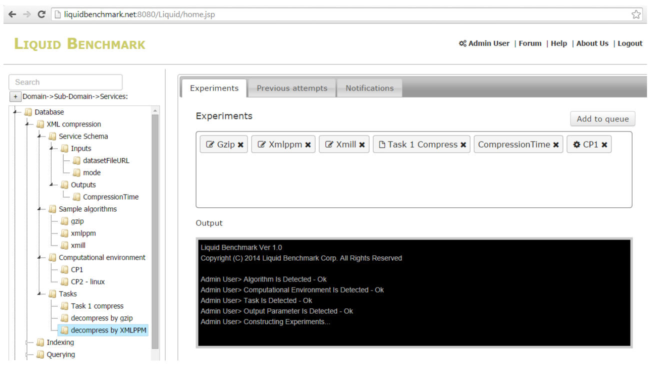


Figure 1: Platfrom Architecture

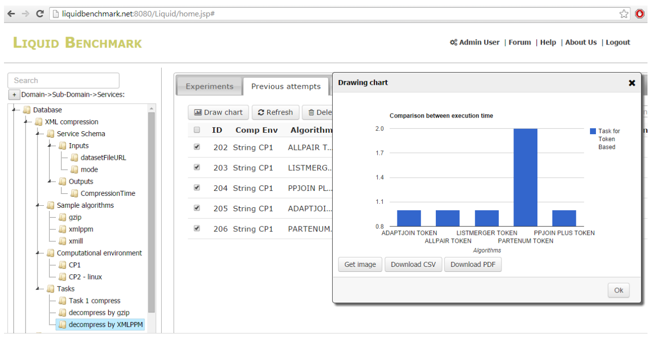
- **Repository of Experiment Results**: This repository stores the results of all experiments associated with their configuration parameters, *provenance* information (e.g. timestamp, user) and social information (e.g. comments, discussions). Clearly, end-users can search and view the contents of this repository to analyze, compare, visualize and comment on the results of the previously running experiments without taking the time of re-running or creating them from scratch.
- **Visualization Manager**: This component is equipped with a set of *visualization styles* (e.g. column charts, line charts) for presenting and comparing the results (metrics) of the selected experiments by the end-user.



```
Liquid Benchmark Ver 1.0
Copyright (C) 2014 Liquid Benchmark Corp. All Rights Reserved

Admin User> Algorithm Is Detected - Ok
Admin User> Computational Environment Is Detected - Ok
Admin User> Task Is Detected - Ok
Admin User> Output Parameter Is Detected - Ok
Admin User> Constructing Experiments...
```

Figure 2: Screenshot: Mashing Up an Experiment



	ID	Comp Env	Algorithm
✓	202	String CP1	ALLPAIR T...
✓	203	String CP1	LISTMERG...
✓	204	String CP1	PPJOIN PL...
✓	205	String CP1	ADAPTJOI...
✓	206	String CP1	PARTENUM...

Figure 3: Screenshot: Comparing and Visualizing Experimental Results

3. DEMONSTRATION SCENARIOS

In this demonstration, we will start by presenting the different features of the Liquid Benchmarking platform[9] such as the process of mashing up a new experiment (Figure 2) or visualizing the experimental results (Figure 3). Then, the demonstration will present four benchmarking case studies that have been implemented using the Liquid Benchmarking platform on the following domains:

- *XML compression*[10]: This case study is based on the benchmark of XML compressors that has been presented in [7]. In particular, this case study provides services for the implementation of nine XML compression tools with benchmarking tasks over an XML corpus that contains 57 documents which are covering the different types and scales of XML documents. This case study evaluates the XML compressors by three different metrics: compression ratio, compression time and decompression time.
- *Graph indexing and querying*[11]: This case study implements the *iGraph* framework [3] for evaluating the graph indexing and querying techniques. In particular, the case study provides the services of seven techniques and evaluates them on the basis of their indexing time, index size and query processing time using a real AIDS antiviral screen dataset (NCI/NIH) and synthetically generated datasets.
- *String Similarity Join*[12]: An implementation for the recent evaluation and comparison study which is presented by Jiang et al. [4]. The case study provides the implementation of twelve algorithms and provides six different experimental datasets. The evaluation of the benchmarked algorithms is based on two metrics: the running time and the size of candidate results.

The case studies of our demonstration will be deployed in two cloud environments: the Amazon public cloud environment[13] in addition to our own private cloud environment which is managed by the *OpenStack* platform[14]. In addition, each case study will be demonstrated using two different testing environments (virtual machines): The first environment will be configured with high computing resources while the other environment will be conifgured with limited computing resources. Furthermore, we will present how the authenticated users can access different services of the platform (e.g., creating and running experiments, searching the repository of results) using its supported RESTful interfaces and API-based SDK[15].

[9]The platform can be accessed online on `http://liquidbenchmark.net:8080/Liquid/`. The full documentation for using the platform is available on `http://wiki.liquidbenchmark.net/`

[10]`http://wiki.liquidbenchmark.net/doku.php/casestudy-xmlcompression`

[11]`http://wiki.liquidbenchmark.net/doku.php/casestudy-graph-indexing-querying`

[12]`http://wiki.liquidbenchmark.net/doku.php/casestudy-string-similarity-join`

[13]`http://aws.amazon.com/`

Acknowledgement

The Liquid Benchmark Project is supported by King Abdulaziz City for Science and Technology (KACST), project 11-INF1992-03.

4. REFERENCES

[1] F. Chirigati, M. Troyer, D. Shasha, and J. Freire. A Computational Reproducibility Benchmark. *IEEE Data Eng. Bull.*, 36(4), 2013.

[2] J. Freire, P. Bonnet, and D. Shasha. Computational reproducibility: state-of-the-art, challenges, and database research opportunities. In *SIGMOD*, 2012.

[3] W. Han, J. Lee, M. Pham, and J. Xu Yu. iGraph: A Framework for Comparisons of Disk-Based Graph Indexing Techniques. *PVLDB*, 3(1), 2010.

[4] Y. Jiang, G. Li, J. Feng, and W. Li. String Similarity Joins: An Experimental Evaluation. *PVLDB*, 7(8), 2014.

[5] S. Manegold and I. Manolescu. Performance evaluation in database research: principles and experience. In *EDBT*, 2009.

[6] I. Manolescu, L. Afanasiev, A. Arion, J. Dittrich, S. Manegold, N. Polyzotis, K. Schnaitter, P. Senellart, S. Zoupanos, and D. Shasha. The repeatability experiment of SIGMOD 2008. *SIGMOD Record*, 37(1), 2008.

[7] S. Sakr. XML compression techniques: A survey and comparison. *JCSS*, 75(5):303–322, 2009.

[8] S. Sakr and F. Casati. Liquid Benchmarks: Towards An Online Platform for Collaborative Assessment of Computer Science Research Results. In *TPCTC*, 2010.

[14]`http://www.openstack.org/`

[15]`http://wiki.liquidbenchmark.net/doku.php/RESTful-interface`

Natural Language Specification and Violation Reporting of Business Rules over ER-modeled Databases

Mika Cohen
FOI, Stockholm, Sweden
mika.cohen@foi.se

Michael Minock
KTH Royal Institute of Technology, Stockholm
minock@kth.se

Daniel Oskarsson
FOI, Stockholm, Sweden
daniel.oskarsson@foi.se

Björn Pelzer
FOI,Stockholm, Sweden
bjorn.pelzer@foi.se

ABSTRACT

This paper presents our work on adapting and extending *natural language interface (NLI) to database* technology to support the specification and violation reporting of business rules. The resulting system allows non-technical users to author and manage a rulebook in controlled natural language – serving as a single point of definition that can be compiled into SQL to generate violation reports. To achieve this we represent business rules in tuple calculus, handle negation in our query re-writing algorithms and add support for natural language reflexives (e.g. 'its', 'themselves', etc.). Our results show a large class of business rules can be captured with these extensions. Although our approach is general, we present it applied to compliance checking of regulations over a materiel capability development information system at the Swedish Defence Materiel Administration. At EDBT we will also demonstrate this work over a more generic package delivery domain. While there has been recent effort in pursuing *Semantics for Business Vocabulary and Business Rules* (SBVR) in the semantic web and description logic communities, to our knowledge ours is the first attempt to provide this capability for ER-modeled relational databases.

Keywords

SBVR, Business Rules, NLIs to databases

1. INTRODUCTION

Large organizations typically maintain a wide range of information systems each with their own interfaces and schemas. Much effort in recent years has been expended to aggregate overall information systems into federated databases (or *enterprise information systems*) so that overall activities can be monitored, analyzed and, if required, remedied. While an individual department can often ensure that its data sources conform to its own rules, violations of organization wide rules often occur when all the sources are federated. For example while it is not be permitted for a clerk to enter an order with no purchase item specified, other compliance level 'constraints', such as ensuring that those who ordered an item have authority to use it, are difficult to enforce in legacy systems. Even at the local level, there may be reasons not to enforce constraints too strictly, such as to monitor states that are not outright errors but nonetheless deviate from expectations in noteworthy ways.

An approach to relaxing constraint enforcement to accommodate these cases is to check for compliance *retroactively*. Operational staff are given the rights to enter data, and then, during *compliance checking*, rules are checked against the federated database, looking for violations of regulations, incompleteness in data sources, breakdowns in operations, etc. With such violation reports, errors may either be corrected, or, based on discretion, tolerated. This temporal decoupling of compliance checking from data entry also allows for a wider spectrum of rule specificity, enabling vaguer, high-level business rules in addition to specific data-level constraints. Rules will then originate from stakeholders at various levels across the organization who build up *rulebooks* in natural language, which are subsequently mapped to low-level, machine-executable implementations, such as SQL.

The translation of natural language rules to executable form can be tedious and costly, and since it is typically done by technical staff that are separate from the rule authors (both organizationally and in terms of skills), the translation involves interpretation – a source of possible errors, especially in light of the ambiguity of natural language. Also, keeping parallel representations (natural language and executable code) of the rulebook introduces a burden of management to maintain consistency over time. Finally, if the natural language formulations are overly verbose, stakeholders in more remote parts of the organization will find it difficult and burdensome to understand the rules.

Based on these shortcomings, recent efforts have focused on *Semantics for Business Vocabulary and Business Rules* (SBVR) [8]. SBVR argues that the specification of business rules should be based on a clear conceptual model of the domain. Moreover the business rules and the conceptual domain must be based on a controlled natural language syntax, so as to reduce or eliminate ambiguity, increasing the probability that all the stakeholders will understand the rules. Finally there is the potential that natural language rules may be automatically mapped to executable form.

10.5441/002/edbt.2015.53

While SBVR is now an OMG standard, implementation work is still in its infancy (see [1] for references to current efforts). Most, if not all, of the tools are focused on realizing SBVR over description logics and semantic web technology (for example [9, 3]). Approaches mapping all the way to SQL are promising, but still quite preliminary[7]. Our work seeks to seat SBVR[1] in classical ER-modeled relational databases. Given the dearth of existing tools, to achieve this, we decided to adapt and extend an existing natural language interfaces to database system, C-Phrase[6] `https://code.google.com/p/c-phrase/`.

In section 2 we introduce the domain in which we are applying our work work – The SKTS at FMV, the Swedish Defence Materiel Administration. Section 2 gives an ER model and several prototypical business rules drawn from our corpus, which we can interpret, paraphrase and report the violations of. Section 3 discusses the extensions of C-Phrase that were required to achieve this. Beyond representing rules, we were required to extend our treatment of negation in paraphrases and were forced to extend the system to support reflexives. Section 4 gives our demonstration plan for EDBT. Finally section 5 discusses the broader relevance and future directions of our work.

2. THE SKTS AT FMV

Let us consider the simple real-world example, the SKTS[2], which is a description of the organization of technical systems (i.e. equipment of non-trivial technical complexity) by the Swedish Armed Forces and managed by the Swedish Defence Materiel Administration FMV (*Försvarets materielverk*). SKTS describes the relations between systems as well as their deployment, their life-cycle phases and the associated decision making processes. Our non-classified extract of SKTS has the same schema as the full version. See Figure 1 for a simplified ER model of the SKTS.

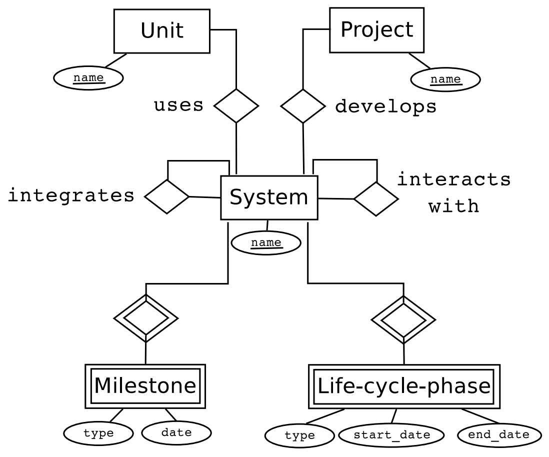


Figure 1: Simplified ER model of SKTS

The central entity in SKTS is the *system*, representing about 1,600 types of technical systems, ranging from ammunition and tools to vehicles and buildings. In our simplified schema a system type is identified by a single attribute *name*. Two special relationships between systems are:

- system type S_1 *integrates* system type S_2: S_2 is a component of S_1, like a tank integrating an engine;
- system type S_1 *interacts with* system type S_2: S_1 cooperates with S_2 (without integration), like artillery interacting with a forward observer vehicle to obtain target data;

The *unit* and *project* entities represent the actors employing technical systems, i.e. military units during active use and research projects during the earlier life-cycle phases. Systems are assigned to such users via the respective relationships *uses* and *develops*.

The usage of systems is constrained by weak entity sets life-cycle phases and milestones. A *life-cycle phase* represents an interval, and thus it has attributes specifying its *start date* and *end date*, and its *type* attribute is one of *concept*, *development*, *production*, *use*, *maintenance* or *retirement*. A system can have multiple life-cycle phases associated with it – ideally one of each type. A *milestone* represents an event like a deadline, so it has only a single *date* attribute. Numerous values are possible for its *type* attribute, since up to three different milestones govern each life-cycle phase of a system: a *deadline* by which the FMV must have decided on the start and end dates of the phase, a *planned decision* date when the FMV intends to make this decision, and a *decision* date when the decision is actually made.

The FMV wants to apply SBVR-formulated business rules to an SKTS-database to ensure that its regulations are not violated within the organization. Five prototypical rules drawn from our initial corpus of hundreds of rules are:

1 "It is forbidden that a system has a use phase that begins before its use decision."

2 "It is obligatory that a system that is in a use phase be used by some unit."

3 "It is obligatory that a system that interacts with another system with a use phase must also have a use phase."

4 "Every system that is assigned to some unit must be currently operational."

5 "It is obligatory that a life-cycle phase which is not a concept phase and which has a start date no later than the current date be subsequent to some other life-cycle phase."

Applying such rules to SKTS should produce a detailed list of violations, allowing an analyst to identify the best method to handle each problem – be it a simple database flaw or an actual breach within the organization, like a unit employing a system not cleared for use.

3. EXTENDING C-PHRASE TO SBVR

We have extended the C-Phrase NLI to database system to enable users to state business rules of the form above (as well as many other rules and alternative phrasings), receive paraphrases of such business rules (in case of ambiguity),

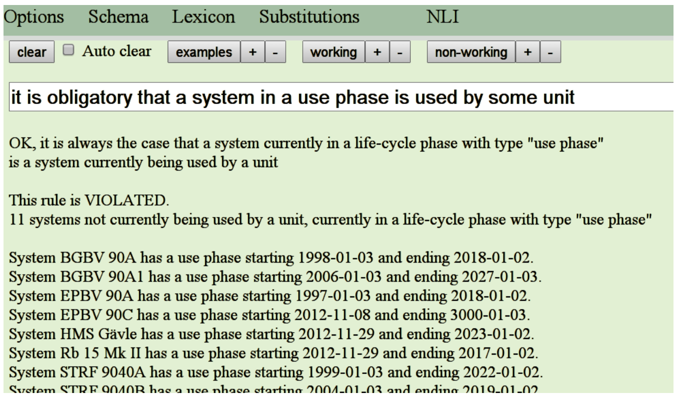
[1]Technically it is SBVR-SE (SBVR Structured English), a subset of SBVR, that we seek to support.
[2]*Systemkarta tekniska system*, Swedish for "system map (of) technical systems"

and receive reports of the violations of such rules that give an indication of how an instance violated the rule. C-Phrase is discussed elsewhere [6], but in short it uses a semantic grammar to map user utterances to (an extended) tuple calculus. From this representation either a natural language paraphrase or SQL may be generated. C-Phrase uses a theorem prover to evaluate when a natural language utterance has mapped to more than one semantically distinct query. In such a case of ambiguity, the system paraphrases all semantically distinct interpretations back to the user so that they may select the proper interpretation. The paraphrasing technique also heavily uses a theorem prover in finding an equivalent re-writing of the paraphrased query using a lexicon of elementary query expressions paired with associated words and phrases. Once a single interpretation of the user's utterance is determined, the tuple calculus expression is evaluated over the database and answers are reported back to the user.

For C-Phrase to handle business rules, several extensions were required. First, it was necessary to extend the tuple calculus representation to represent rules. This was only necessary in the case of positive rules. *Negative rules* expressed as "it is prohibited that A", may simply be represented as the tuple expressions for A, leveraging C-Phrase's existing semantic analysis mechanism. For example, the first example rule above is represented in the tuple query expression[3]: $\{x|System(x) \wedge (\exists y_1)(Life\text{-}Cycle\text{-}Phase(y_1) \wedge x.name = y_1.system \wedge y_1.type = \text{'use phase'} \wedge (\exists y_2)(milestone(y_2) \wedge y_1.start\text{-}date < y_2.date \wedge (\exists y_3)\ (System(y_3) \wedge y2.system = y_3.name \wedge \mathbf{x} = \mathbf{y_3}\ \wedge y_3.type = \text{'use start decision'})))\}$. For *positive rules*, expressed as "it is necessary that A are B", we introduced a **:consequent** marker, which specifies which part of a tuple calculus expression is the right hand side of a rule. For example the third rule above is represented as: $\{x|System(x)(\exists y_1)(\exists y_2)(Interacts\text{-}With(y_1) \wedge System(y_2) \wedge x.name = y_1.interacts\text{-}with \wedge y_1.system = y_2.name \wedge (\exists y_3)(Life\text{-}Cycle\text{-}Phase(y_3) \wedge y_2.name = y_3.system \wedge y_3.type = \text{'use phase'}))\langle : \textbf{consequent}\ System(x) \wedge (\exists y_4)(Life\text{-}Cycle\text{-}Phase(y_4) \wedge x.name = y_4.system \wedge y_4.type = \text{'use phase'})\rangle\}$. Finally, C-Phrase's grammar was extended to recognize a large set of variations of "it is prohibited that" (or "it is obligatory that"), to insert the **:consequent** marker in the positive case, and then to branch to the proper *rule handler* with the resulting tuple calculus expression.

Although our semantic grammar avoids the pathological ambiguity of more linguistically-oriented approaches (e.g. "time flies like an arrow"), we do still confront ambiguity. For example in rule specification 5 above, there is a hidden ambiguity in 'subsequent'. 'Subsequent' in this domain may mean immediately subsequent. Or subsequent with a possible time gap. There are thus two semantically distinct rules that rule 5 above maps to. Thus the user must pick between them. And to pick, the user must understand the nuances in these formulas. And for that, among other reasons, we paraphrase rules back to the user in natural language. At the end of this interaction, we will have one single semantically meaningful rule to pass to the rule handler.

The rule handler converts the rule into one or more *violation queries* which identify violation cases for the rule. This occurs in a two step process. First, a *core violation query* is calculated. Then this core violation query is extended to the (full)*violation query*, which includes supporting information indicating why matching answers to the core violation query are rule violators. In the negative case, the core violation query is simply the query representing the rule. In the positive case, the core violation query is the query with the consequent negated. In cases where the consequent is a conjunction, we apply De Morgan's law, followed by a simple rewriting to generate a set of core violation queries. Extending core violation queries to full violation queries is a capability inherited from C-Phrase's answer strategy mechanism. In short, the theorem prover finds the most specific answer strategy that subsumes the core violation query and then augments the core violation query with the associated answer value bindings. For example, assume that the lexicon contains the answer strategy $\langle\{x|System(x) \wedge x.name = c_1 \wedge (\exists y_1)(Life\text{-}Cycle\text{-}Phase(y_1) \wedge y_1.system = x.name \wedge y_1.type = \text{'use phase'} \wedge y_1.start\text{-}date = c_2 \wedge y_1.start\text{-}date = c_3)\}$:*"The system c_1 has a use phase starting c_2 and ending c_3"*$\rangle$. Then a core violation query that is subsumed by this answer strategy, and by no more specific answer strategy, will be augmented with the additional bindings, and an answer fitting the template will be generated (The reported answers in Figure 2 apply this answer template). The translation of the full violation query to SQL is trivial. The report presented to the user consists of a paraphrase of the rule, followed by a paraphrase of the core violation query, followed by the answers to the full violation query. The report for rule 2 above is shown in Figure 2.



it is obligatory that a system in a use phase is used by some unit

OK, it is always the case that a system currently in a life-cycle phase with type "use phase"
is a system currently being used by a unit

This rule is VIOLATED.
11 systems not currently being used by a unit, currently in a life-cycle phase with type "use phase"

System BGBV 90A has a use phase starting 1998-01-03 and ending 2018-01-02.
System BGBV 90A1 has a use phase starting 2006-01-03 and ending 2027-01-03.
System EPBV 90A has a use phase starting 1997-01-03 and ending 2018-01-02.
System EPBV 90C has a use phase starting 2012-11-08 and ending 3000-01-03.
System HMS Gävle has a use phase starting 2012-11-29 and ending 2023-01-02.
System Rb 15 Mk II has a use phase starting 2012-11-29 and ending 2017-01-02.
System STRF 9040A has a use phase starting 1999-01-03 and ending 2022-01-02.
System STRF 9040B has a use phase starting 2004-01-03 and ending 2019-01-02

Figure 2: Report for rule 2.

Given that paraphrases of rules or violation queries necessarily contain negation of existential quantification, and given that our techniques of generating paraphrases involves finding equivalent re-writings of queries using elementary logical expressions, we were forced to confront a difficult non-conjunctive query rewriting problem. As a simple example, consider that the expression $\{x|System(x) \wedge \neg(\exists y_1)(\exists y_2)(Interacts\text{-}with(y_1) \wedge Sytem(y_2) \wedge y_1.system = x.name \wedge y1.interacts\text{-}with = y_2.name \wedge y_2.name = \text{"HMS Gavle"})\}$ should be rewritten using the three lexicon entries $\langle\{x|System(x)\}$:*"systems"*$\rangle$, $\langle\{x|System(x) \wedge \neg(\exists y_1)(\exists y_2)(Interacts\text{-}with(y_1) \wedge Sytem(y_2) \wedge y_1.system = x.name \wedge y1.interacts\text{-}with = y_2.name \wedge \varphi(y_2))\}$:*"does not interact with ref($\varphi(y_2)$)"*$\rangle$ and $\langle\{x|System(x) \wedge x.name = c_1\}$:"c_1"$\rangle$ to yield an equivalent rewrite, and in turn the

[3]Queries are defined over the schema that is the standard translation of the ER model in figure 1.

templates may be used to generate the paraphrase *"systems not interacting with HMS Gavle"*. Although we have not yet formalized our approach to the point where we can prove its completeness, we have considerably extended our query rewriting algorithms to handle negation.

A final extension to C-Phrase that should be noted is support for reflexives. Normally in NLIs to databases one does not confront reflexives. A contrived example could be, "give the systems interacting with *themselves*". In business rules however, we have witnessed a fair number of cases in our corpus that require this. For example the *'its'* in the first rule above is a reflexive. Our approach to reflexives is similar to that for **:consequent**. We insert a marker **:reflexive** in at parse time when reflexives are recognized. This leads to a tuple calculus expression which is later resolved, finding bindings of the reflexive variables with the same entity type. The semantic representation of rule 1 presented above shows the results of this with the tuple equality condition $\mathbf{x} = \mathbf{y_3}$. In cases of multiple possibilities, all possibilities are passed on for interactive ambiguity resolution. The paraphrasing mechanism was trivially extended to generate reflexives.

4. DEMONSTRATION

Our demonstration plan at EDBT is to first show the live operation of the system in interactive mode, where a user states business rules over the SKTS schema above, receives paraphrases of rules, and then receives a detailed report on the violations of the rules over our declassified database instance. We will demonstrate all the rules in section 3 as well as additional rules in our corpus. We will also present rules that still give us problems due to either linguistic or conceptual complexity. In addition to the live demonstration, we will generate a series of videos that illustrates how we configured C-Phrase to handle this task. Moreover we will demonstrate the same technology over a prototypical use case in the package delivery domain. All configuration files and source code for our demonstration will be open sourced to let others verify, replicate and build on our results.

5. RELEVANCE AND CONCLUSIONS

It has been noted many times [2, 6, 4, 5] that natural language interfaces to critical systems must include a paraphrasing mechanism which communicates back to the user how their utterance is interpreted. Without such a mechanism how could anyone ever rely upon such a system? Without such a capability how would one let users resolve ambiguity? This is particularly true with the interpretation and reporting of business rules which, arguably, are more complex than information seeking queries. Given that paraphrasing is critical, and that even very simple business rules may be reported in either positive or negative forms, interpretation and paraphrasing mechanism must be extended to handle negation over existential quantifiers; NLI systems restricted to conjunctive query representations, will simply not suffice. Our first contribution in this work is to demonstrate that such negation may be built into NLI systems to adequately handle this requirement. In addition we observed in our work the importance of supporting reflexives in the specification of business rules; others attempting to achieve the same will likewise need to cover reflexives. Finally we argue that the use of a theorem prover is critical to determining semantic equivalence and greatly simplifies the proper reporting of violations of business rules.

There are still types of business rules in our corpus that we do not cover. For example cardinality type queries are not yet covered. Nor do we yet support offsets in time expressions. So the rule, *"It is prohibited that a project develops a system that interacts with at least three other systems whose retirement phases begin less than two years after the start date of its use phase."* is currently beyond our grasp and will require both syntactic and semantic extensions. These extensions are currently being explored. Moreover, we are investigating the building up of complex rules, and rules with exceptions, as multiple step interactions. In some domains, sufficiently detailed rules can probably not be specified as single shot sentences.

While it is necessary to configure C-Phrase over the ER modeled database, one side benefit is that one gets an NLI access interface as an added bonus, justifying in part the additional cost. Moreover, because a C-Phrase configuration can be tailored by database administrators with limited linguistic training, the configuration may be extended to capture domain dependent idiosyncratic phrasings. As database administration staff build out the natural language interface over the ER modeled database, we envision stakeholders engaging in the process of proposing and organizing business rules that may be shared, understood and most importantly executed across the organization. Our work takes a step toward realizing this vision.

6. REFERENCES

[1] I. S. Bajwa, M. G. Lee, and B. Bordbar. SBVR business rules generation from natural language specification. In *AI for Business Agility, AAAI Spring Symposium*, 2011.

[2] E. Codd. Seven steps to rendezvous with the casual user. In *IFIP Working Conference Data Base Management*, pages 179–200, 1974.

[3] C. Fürber and M. Hepp. Towards a vocabulary for data quality management in semantic web architectures. In *Proceedings of the 2011 EDBT/ICDT Workshop on Linked Web Data Management, Uppsala, Sweden, March 25, 2011*, pages 1–8, 2011.

[4] G. Koutrika, A. Simitsis, and Y. E. Ioannidis. Explaining structured queries in natural language. In *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, pages 333–344, 2010.

[5] F. Li and H. V. Jagadish. Nalir: an interactive natural language interface for querying relational databases. In *Proc. of SIGMOD 2014*, pages 709–712, 2014.

[6] M. Minock. A STEP towards realizing Codd's vision of rendezvous with the casual user. In *Proc. of Very Large Data Bases (VLDB)*, pages 1358–1361, 2007.

[7] S. Moschoyiannis, A. Marinos, and P. Krause. Generating SQL queries from SBVR rules. In *Semantic Web Rules*, volume 6403 of *LNCS*, pages 128–143. Springer, 2010.

[8] OMG. *Semantics of Business Vocabulary and Rules (SBVR) (version 1.2)*. November 2013.

[9] D. Solomakhin, E. Franconi, and A. Mosca. Logic-based reasoning support for SBVR. In *Proc. of the 26th Italian Conference on Computational Logic*, pages 311–325, 2011.

POIESIS: a Tool for Quality-aware ETL Process Redesign

Vasileios Theodorou[1] Alberto Abelló[1] Maik Thiele[2] Wolfgang Lehner[2]

[1]Universitat Politécnica de Catalunya
Barcelona, Spain
{vasileios,aabello}@essi.upc.edu

[2] Technische Universität Dresden
Dresden, Germany
{maik.thiele,wolfgang.lehner}@tu-dresden.de

ABSTRACT

We present a tool, called **POIESIS**, for automatic ETL process enhancement. ETL processes are essential data-centric activities in modern business intelligence environments and they need to be examined through a viewpoint that concerns their quality characteristics (e.g., data quality, performance, manageability) in the era of Big Data. **POIESIS** responds to this need by providing a user-centered environment for quality-aware analysis and redesign of ETL flows. It generates thousands of alternative flows by adding flow patterns to the initial flow, in varying positions and combinations, thus creating alternative design options in a multidimensional space of different quality attributes. Through the demonstration of **POIESIS** we introduce the tool's capabilities and highlight its efficiency, usability and modifiability, thanks to its polymorphic design.

1. INTRODUCTION

The increasing volume of available data, as well as the requirement for recording and responding to multiple events coming from participants within Big Data ecosystems that are characterized by the 3Vs (volume, variety, velocity) [3], pose a serious challenge for modern data-centric processes. As such, Extract-Transform-Load (ETL) processes are becoming more and more complex, while there is a growing demand for their real time responsiveness and user-centricity.

It has recently been proposed that to tackle complexity, the level of abstraction for ETL processes can be raised. ETL processes have been decomposed to ETL activities [6] and recurring patterns [1] as the main elements of their workflow representation, making them susceptible to analysis for process evaluation and redesign.

Manual modification of ETL processes in order to improve their quality characteristics is error-prone, non-trivial, time-consuming and it suffers from incompleteness, inefficiency, and ineffectiveness. According to our experience with individuals with computer science expertise, most common mistakes during this manual process are wrong configuration of ETL operations, incomplete exploitation of quality enhancement options and wrong placement of optimization patterns.

It is apparent that there is a need for an automatized process of ETL quality enhancement, as it would solve many of the above-mentioned issues. Analysts should be in the center of this process, where the large problem space is automatically generated, simulated and displayed in an intuitive representation, allowing for the selection among alternative design choices.

In this paper we present our tool **POIESIS**, which stands for **P**rocess **O**ptimization and **I**mprovement for **E**TL **S**ystems and **I**ntegration **S**ervices. Using a process perspective of an ETL activity, our tool can improve the quality of an ETL Process by automatically generating optimization patterns integrated in the ETL flow, resulting to thousands of alternative ETL flows. We apply an iterative model where users are the key participants through well-defined collaborative interfaces and based on estimated measures for different quality characteristics. POIESIS implements a modular architecture that employs reuse of components and patterns to streamline the design. Our tool can be used for incremental, quantitative improvement of ETL process models, promoting automation and reducing complexity. Through the automatic generation of alternative ETL flows, it simplifies the exploration of the problem space and it enables further analysis and identification of correlations among design choices and quality characteristics of the ETL models.

The remainder of this paper is organized as follows: In Section 2 we provide some background for ETL quality analysis and redesign; in Section 3 we provide an overview of the system and finally, in Section 4 we showcase an outline of a demonstration of our tool.

2. QUALITY-AWARE REDESIGN OF ETL

2.1 ETL Quality Characteristics

ETL processes need to be evaluated in a scope that brings them closer to fitness to use for data scientists. Therefore, apart from performance and cost, other quality characteristics, as well as the trade-offs among them should be taken under consideration during ETL analysis. In Fig. 6 we show a subset of ETL process quality characteristics and measures that we have extracted from existing literature, in our previous work [4]. There are two types of measures: ones that derive directly from the static structure of the process model

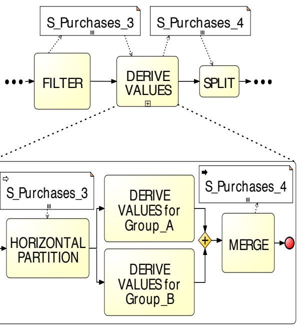
10.5441/002/edbt.2015.54

and those that are obtained from analysis of historical traces capturing the runtime behaviour of ETL components.

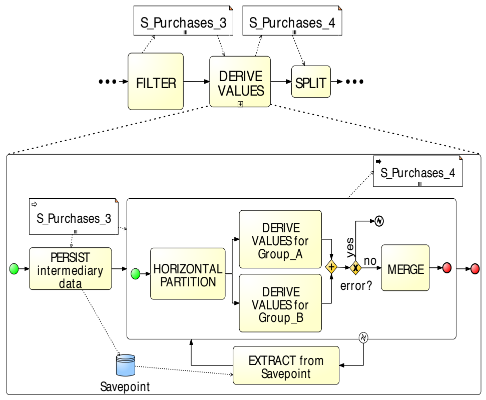
Characteristic	Measure
performance	• Process cycle time • Average latency per tuple
data quality	• Request time - Time of last update • 1 / (1 - age * Frequency of updates)
manageability	• Length of process workflow's longest path • Coupling of process workflow • # of merge elements in the process model

Figure 1: Example quality measures for ETL processes

Based on such measures, it is possible to conduct a multi-objective analysis and make design decisions according to user preferences on different quality characteristics, which can often be conflicting.



(a) Improved performance



(b) Improved reliability

Figure 2: Generation of FCP on the ETL flow

2.2 Addition of Flow Component Patterns

An initial ETL flow can be modified with the addition of predefined constructs that improve certain quality characteristics, but do not alter its main functionality. We refer to these constructs as Flow Component Patterns (FCP) and their integration can take place on different parts of the initial flow, depending on the flow topology. For example, in Fig. 2, we illustrate how different quality goals can cause the generation of different FCP on the ETL flow. In the first case, the goal of improving time performance of the process, results in the generation of horizontal partitioning and parallelism within a computational-intensive task and in the second, the goal of improving reliability brings about the addition of a recovery point to the sub-process. Another example would be the goal of improved data quality that would result in crosschecking with alternative data sources.

Central to our implementation is the notion of *application point* of a FCP, which can be either a node (i.e., an ETL flow operation), or an edge or the entire ETL flow graph. As examples, a valid application point for the *ParallelizeTask* pattern is a node that can be replaced by multiple copies of itself and a valid application point for the *FilterNullValues* pattern is an edge on which a filter operation can be added. The entire ETL flow graph as application point serves for the case of process-wide configuration and management operations that are not directly related to the functionality of specific flow components. Examples of the latter include the application of security configurations (encryption, role-based access etc.), management of the quality of Hw/Sw resources, adjusting the frequency of process recurrence etc.

We model the ETL process as one graph G with graph components (V, E), where each node (V) represents an ETL flow operation, and each edge (E) represents a transition from one operation to a successor one. We also assume that there is a set P of available FCP, $P = P_E \cup P_V \cup P_G$, each of which can either be applied on a node, an edge of G, or the entire graph, in order to improve one or more quality characteristics of the ETL flow.

After the application of all the FCP, a number of nodes and edges is added to the initial graph. This process can be repeated an arbitrary number of times and a new Graph is created every time.

It is apparent that the complexity of this analysis is factorial to the size of the graph. Thus, manual configuration of the ETL flow appears inefficient and error-prone, being dependent not only on the users' cognitive abilities but also on characteristics and dynamics of the flow that are hard to predict. Therefore, the need for defining adequate automated mechanisms and heuristics to produce and explore alternative designs and to optimize the ETL flow is evident.

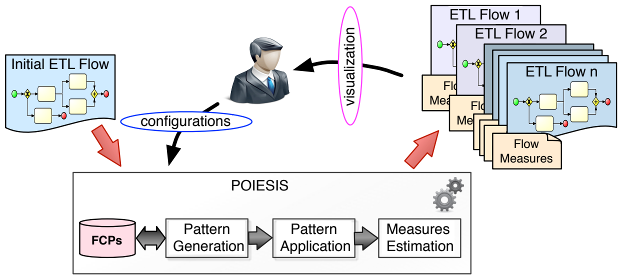


Figure 3: POIESIS architecture

3. SYSTEM OVERVIEW

In [5] we have presented an architecture for user-centered, declarative ETL (re-)design. **POIESIS** is an implementation of the *Planner* component of that architecture. The main functionality of this component is the automatic application of Flow Component Patterns (FCP) on an existing ETL process flow and the architecture of our approach can be seen in Fig. 3. **POIESIS** takes as input an initial ETL flow and user-defined configurations. Utilizing an existing repository of FCP models, it generates patterns that are specific to the ETL flow on which they are applied. Thus, it produces alternative ETL designs with different FCPs and varying distribution of them on the ETL flow, while keeping the data sources schemata constant. It also estimates defined measures for various quality attributes and illustrates the alternative flows, as well as the corresponding measures to the user through an intuitive visualization.

The internal representation of the FCPs is in the same format as the process flow on which they are deployed. Thus, they can be considered as additional flow components which are positioned at valid application points of the process flow. For example, the *FilterNullValues* pattern is itself an ETL flow consisting of only one operation — a filter that deletes entries with null values from its input. When the *FilterNullValues* pattern is deployed on the initial ETL flow, it is interposed between two consecutive operations. The *FilterNullValues* ETL flow is then configured according to the properties and characteristics of the initial ETL flow as well as the exact application point, ensuring the consistency between data schemata, run-time parameters etc. The same idea is generalized for more complex FCPs or for their more elaborate implementations (e.g., data enrichment additionally to data removal in the described example). In those cases, more detailed configurations might be required to be predefined, such as the access points and data models of additional data sources and processing algorithms of operations.

Our main drivers throughout the development of this component have been the objectives of extensibility and efficiency. In this direction, we followed a modular design with clear-cut interfaces and we employed well-known object-oriented design patterns. The model that was used internally to represent the ETL process flow and allow for its modifications was the ETL flow graph. Each node of this graph represents an ETL flow operation and each directed edge represents a transition from one operation to a successor one.

As a consequence, one strong point of our implementation is that it allows for the definition of custom, additional FCPs, tailored to specific use cases. The applicability of a FCP on the complete ETL flow or some part of it, is decided based upon specific conditions that form the applicability prerequisites, such as the presence or not of specific data types in the operation schemata (e.g., numeric fields in the output schema of preceding operator). Each FCP is related to a particular set of prerequisites that have to be satisfied conjunctively to determine a valid application point. Apart from these strict conditions, there are also *heuristics* to determine the fitness of FCPs for different parts of the ETL flow. For example, according to such heuristics, the addition of a checkpoint is encouraged after the execution of the most complex operations of the ETL fow, in order to avoid the repetition of process-intensive tasks in case of a recovery. Similarly, the application of FCPs related to data cleaning is encouraged as close as possible to the operations for inputing data sources, to prevent cumulative side-effects of reduced data quality. Thus, as opposed to manual deployment, our tool guarantees that all of the potential application points on the ETL flow are checked for each FCP and it can be customized to select the deployment of patterns based on custom policies based on different heuristics.

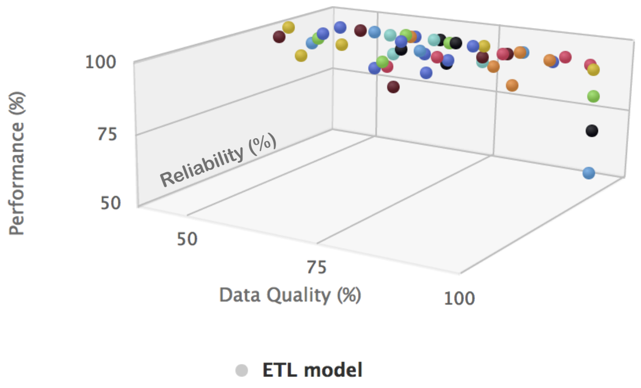


Figure 4: Multidimensional scatter-plot of alternative ETL flows

What is unique about **POIESIS** is that the redesign process takes place in an iterative, incremental and intuitive fashion. A large number of alternative process designs is automatically generated and these can be instantly evaluated based on quality criteria. Moreover, through a highly interactive UI, the user at any point can interact with a visualization of the ETL process and the estimated measures for each of the alternative designs.

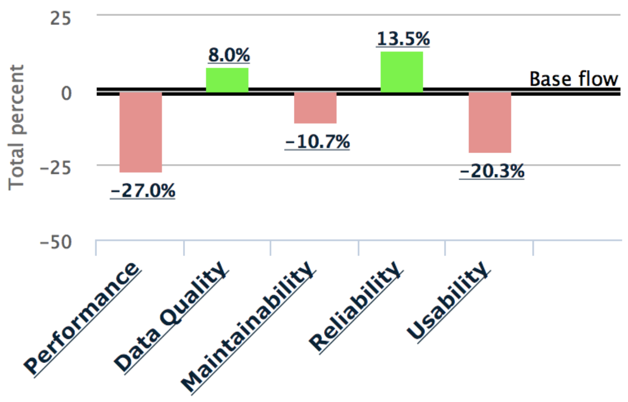


Figure 5: Relative change of measures for an ETL flow, compared with the initial flow as a baseline

The first step is to import an initial ETL model to the system. This model can be a logical representation of the ETL process and we currently support the loading of xLM [7] and PDI[1], but more options will be available in a future version. Subsequently, the user can select the preferred processing parameters, i.e., choose which FCP can be considered in the palette of patterns to be added to the flow, and select the deployment policy for the patterns. It is important to notice at this point that the user can configure the various patterns and even extend them to create custom patterns for future use. The same also stands for the deployment policies, which can be configured according to the user-defined prioritization of goals, as well as the set of constraints based on estimated measures.

Next, after generating and applying relevant FCPs on the ETL flow, the Planner presents to the user a set of potential

[1]`http://community.pentaho.com/projects/data-integration/`

designs in a multidimensional scatter-plot visualization (see Fig. 4), together with quality measures (by clicking on any point on the scatter-plot). The scatter-plot points presented to the user are only the Pareto frontier (skyline) of the complete set of alternative designs, based on their evaluation according to the examined quality dimensions, where larger values are preferred to smaller ones. For example, considering the quality dimensions shown in Fig. 4, for one design *ETL1*, if there exists at least one alternative design *ETL2* offering the same or better *performance* and *data quality*, and at the same time better *reliability*, then *ETL1* will not be presented to the user.

The presented measures (see Fig. 5) show on a bar-graph the relative change on the metrics for each quality characteristic, denoting the estimated effect of selecting each of the available flows, compared with the initial flow. Apparently, the processing and analysis of the alternative process designs is a process intensive task, mainly due to the large number of alternative flows that have to be concurrently evaluated. Therefore, we employ Amazon Cloud[2] elastic infrastructures, by launching processing nodes that run in the background and enable system responsiveness.

When the user selects (clicks on) any of the bars on the measures graph, the corresponding composite measure "expands" to more detailed measures, providing the user with a more in-depth monitoring view. Based on measures and design, the user makes a selection decision and the tool implements this decision by integrating the corresponding patterns to the existing process flow. These patterns are in the form of process components and the Planner carefully merges them to the existing process [2]. Subsequently, new iteration cycles commence, until the user considers that the flow adequately satisfies quality goals [4].

4. DEMO WALKTHROUGH

In the demonstration of **POIESIS** we will use two initial ETL processes based on the TPC-DS[3] and TPC-H[4] benchmarks. These processes contain tens of operators, extracting data from multiple sources. Their logical representation in *xLM* format will be loaded in the system and the automatic addition of Flow Component Patterns in different positions and combinations on the initial flows, will result in thousands of alternative ETL flows, with different quality characteristics.

Using these processes as input data to our system, we will show the capabilities of our tool in an interactive demo, consisting of the following parts:

P1. In the first part of the demo, users will interact with the visualizations of our tool's GUI. In particular, they will be able to scroll over/click on any point on the scatterplot that depicts alternative ETL flows on a multidimensional space of different quality characteristics. By selecting one point — corresponding to one ETL flow — the process representation and the measures for this flow will appear on the screen. Users will then be able to view details about the ETL flow, as well as click on any measure so that it expands to more detailed composing metrics.

P2. The second part aims at illustrating how the processing parameters can be configured in order to produce different collections of alternative flows. Thus, users will be allowed to choose which of the available Flow Component Patterns will be used and which policy will be followed for their deployment.

P3. Finally, users will be guided through defining their own Flow Component Patterns, quality metrics and deployment policies, by extending and pre-configuring the existing ones. They will be able to save their custom processing preferences, adding them to the palette of available patterns for future execution. Examples of the FCPs, which our palette currently includes, together with the quality attribute that they are intended to improve, are as follows:

FCP	**Related quality attribute**
RemoveDuplicateEntries	Data Quality
FilterNullValues	Data Quality
CrosscheckSources	Data Quality
ParallelizeTask	Performance
AddCheckpoint	Reliability

Figure 6: Available FCPs

Acknowledgements. This research has been funded by the European Commission through the Erasmus Mundus Joint Doctorate "Information Technologies for Business Intelligence - Doctoral College" (IT4BI-DC).

References

[1] Castellanos, M., Simitsis, A., Wilkinson, K., Dayal, U.: Automating the loading of business process data warehouses. In: EDBT. pp. 612–623 (2009)

[2] Jovanovic, P., Romero, O., Simitsis, A., Abelló, A.: Integrating ETL Processes from Information Requirements. In: DaWaK. pp. 65–80 (2012)

[3] Russom, P.: TDWI best practices report: Big data analytics. Tech. rep., The data Warehousing Institute (01 2011)

[4] Theodorou, V., Abelló, A., Lehner, W.: Quality Measures for ETL Processes. DaWaK (2014)

[5] Theodorou, V., Abelló, A., Thiele, M., Lehner, W.: A Framework for User-Centered Declarative ETL. In: DOLAP (2014)

[6] Vassiliadis, P., Simitsis, A., Baikousi, E.: A taxonomy of ETL activities. In: DOLAP. pp. 25–32 (2009)

[7] Wilkinson, K., Simitsis, A., Castellanos, M., Dayal, U.: Leveraging business process models for ETL design. In: ER, pp. 15–30 (2010)

[2]`http://aws.amazon.com/ec2/`
[3]`http://www.tpc.org/tpcds/`
[4]`http://www.tpc.org/tpch/`

Quarry: Digging Up the Gems of Your Data Treasury

Petar Jovanovic
Universitat Politècnica de Catalunya, BarcelonaTech
Barcelona, Spain
petar@essi.upc.edu

Oscar Romero
Universitat Politècnica de Catalunya, BarcelonaTech
Barcelona, Spain
oromero@essi.upc.edu

Alkis Simitsis
HP Labs
Palo Alto, CA, USA
alkis@hp.com

Alberto Abelló
Universitat Politècnica de Catalunya, BarcelonaTech
Barcelona, Spain
aabello@essi.upc.edu

Héctor Candón
Universitat Politècnica de Catalunya, BarcelonaTech
Barcelona, Spain
hector.candon@est.fib.upc.edu

Sergi Nadal
Universitat Politècnica de Catalunya, BarcelonaTech
Barcelona, Spain
snadal@essi.upc.edu

ABSTRACT

The design lifecycle of a data warehousing (DW) system is primarily led by requirements of its end-users and the complexity of underlying data sources. The process of designing a multidimensional (MD) schema and back-end extract-transform-load (ETL) processes, is a long-term and mostly manual task. As enterprises shift to more real-time and 'on-the-fly' decision making, business intelligence (BI) systems require automated means for efficiently adapting a physical DW design to frequent changes of business needs. To address this problem, we present *Quarry*, an end-to-end system for assisting users of various technical skills in managing the incremental design and deployment of MD schemata and ETL processes. *Quarry* automates the physical design of a DW system from high-level information requirements. Moreover, *Quarry* provides tools for efficiently accommodating MD schema and ETL process designs to new or changed information needs of its end-users. Finally, Quarry facilitates the deployment of the generated DW design over an extensible list of execution engines. On-site, we will use a variety of examples to show how *Quarry* facilitates the complexity of the DW design lifecycle.

1. INTRODUCTION

Traditionally, the process of designing a multidimensional (MD) schema and back-end extract-transform-load (ETL) flows, is a long-term and mostly manual task. It usually includes several rounds of collecting requirements from end-users, reconciliation, and redesigning until the business needs are finally satisfied. Moreover, in today's BI systems, deployed DW systems, satisfying the current set of requirements is subject to frequent changes as the business evolves. MD schema and ETL process, as other software artifacts, do not lend themselves nicely to evolution events and in general, maintaining them manually is hard. First, for each new, changed, or removed requirement, an updated DW design must go through a series of validation processes to guarantee the *satisfaction* of the current set of requirements, and the *soundness* of the updated design solutions (i.e., meeting MD integrity constraints [9]). Moreover, the proposed design solutions should be further optimized to meet different quality objectives (e.g., performance, fault tolerance, structural complexity). Lastly, complex BI systems may usually involve a plethora of execution platforms, each one specialized for efficiently performing a specific analytical processing. Thus the efficient deployment over different execution systems is an additional challenge.

Translating information requirements into MD schema and ETL process designs has been already studied, and various works propose either manual (e.g., [8]), guided (e.g., [1]) or automated [2, 10, 11] approaches for the design of a DW system. In addition, in [4] a tool (a.k.a. *Clio*) is proposed to automatically generate correspondences (i.e., schema mappings) among different existing schemas, while another tool (a.k.a. *Orchid*) [3] further provides interoperability between *Clio* and procedural ETL tools. However, *Clio* and *Orchid* do not tackle the problem of creating a target schema. Moreover, none of these approaches have dealt with automating the adaptation of a DW design to new information needs of its end-users, or the complete lifecyle of a DW design.

To address these problems, we built *Quarry*, an end-to-end system for assisting users in managing the complexity of the DW design lifecycle.

Quarry starts from high-level information requirements expressed in terms of analytical queries that follow the well-known MD model. That is, having a subject of analysis and its analysis dimensions (e.g., *Analyze the revenue from the last year's sales, per products that are ordered from Spain.*). *Quarry* provides a graphical assistance tool for guiding non-expert users in defining such requirements using a domain-specific vocabulary. Moreover, *Quarry* automates the process of validating each requirement with regard to the MD integrity constraints and its translation into MD schema and ETL process designs (i.e., *partial designs*).

Independently of the way end-users translate their information requirements into the corresponding *partial designs*, *Quarry* provides automated means for integrating these MD schema and ETL process designs into a *unified* DW design satisfying all requirements met so far.

10.5441/002/edbt.2015.55

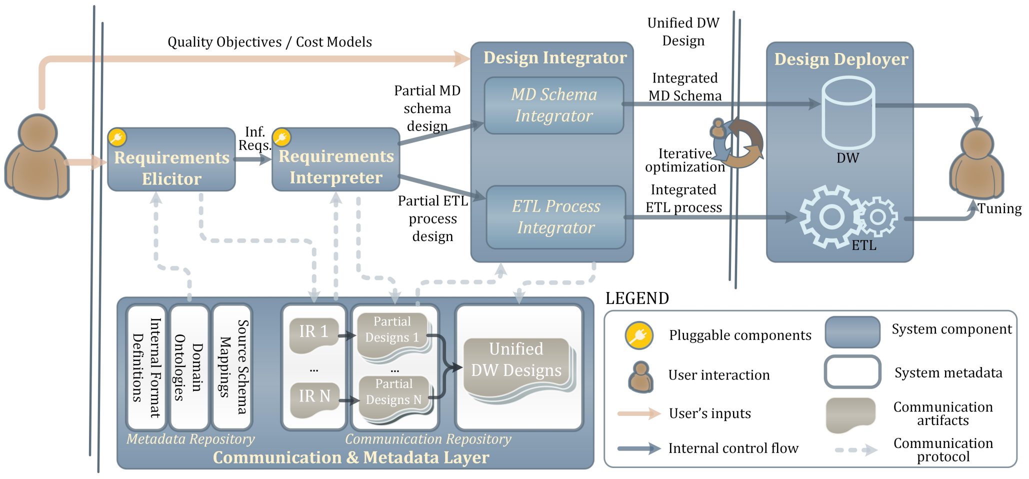


Figure 1: Quarry: system overview

Quarry automates the complex and time-consuming task of the incremental DW design. Moreover, while integrating *partial designs*, *Quarry* provides an automatic validation, both regarding the *soundness* (e.g., meeting MD integrity constraints) and the *satisfiability* of the current business needs. Finally, for leading the automatic integration of MD schema and ETL process designs, and creating an optimal DW design solution, *Quarry* accounts for user-specified quality factors (e.g., *structural design complexity* of an MD schema, *overall execution time* of an ETL process).

Since *Quarry* assists both MD schema and ETL process designs, it also efficiently supports the additional iterative optimization steps of the complete DW design. For example, more complex ETL flows may be required to reduce the complexity of an MD schema and improve the performance of OLAP queries by pre-aggregating and joining source data.

Besides efficiently supporting the traditional DW design, the automation that *Quarry* provides, largely suits the needs of modern BI systems requiring rapid accommodation of a design to satisfy frequent changes.

Outline. We first provide an overview of *Quarry* and then, we present its core features to be demonstrated. Lastly, we outline our on-site presentation.

2. DEMONSTRABLE FEATURES

Quarry presents an end-to-end system for managing the DW design lifecycle. Thus, it comprises four main components (see Figure 1): *Requirements Elicitor*, *Requirements Interpreter*, *Design Integrator*, and *Design Deployer*.

For supporting non-expert users in providing their information requirements at input, *Quarry* provides a graphical component, namely *Requirements Elicitor* (see Figure 2). *Requirements Elicitor* then connects to a component (i.e., *Requirements Interpreter*), which for each information requirement at input semi-automatically generates validated MD schema and ETL process designs (i.e., *partial designs*). *Quarry* further offers a component (i.e., *Design Integrator*) comprising two modules for integrating *partial* MD schema and ETL process designs processed so far, and generating unified design solutions satisfying a complete set of requirements. At each step, after integrating *partial designs* of a new requirement, *Quarry* guarantees the *soundness* of the *unified* design solutions and the *satisfiability* of all re-

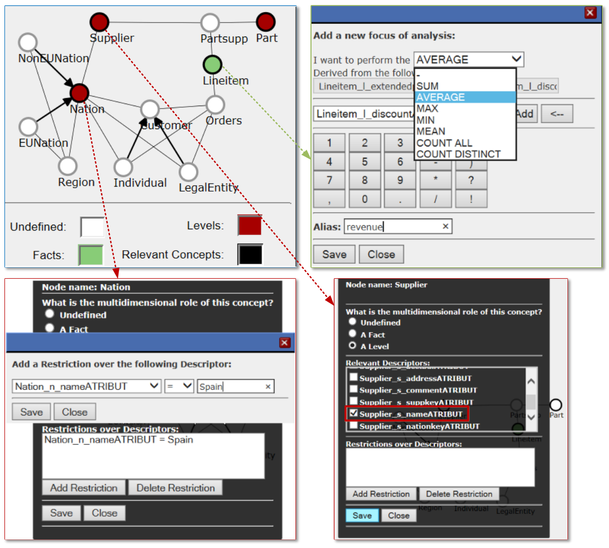


Figure 2: Requirements Elicitor

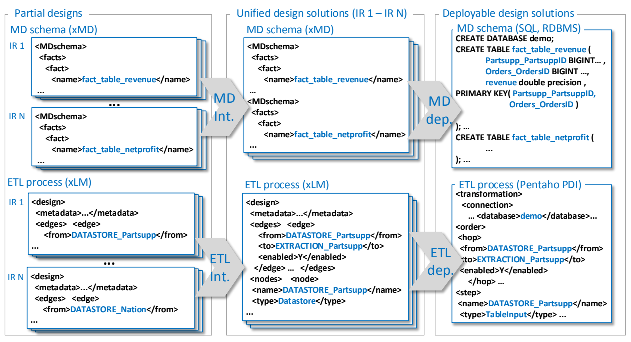
quirements processed so far. The produced DW design solutions are further sent to the *Design Deployer* component for the initial deployment of a DW schema and an ETL process that populates it. The deployed design solutions are then available for further user-preferred tunings and use.

To support intra and cross-platform communication, *Quarry* uses the *communication & metadata* layer (see Figure 1).

2.1 Requirements Elicitor

Requirements Elicitor uses a graphical representation of a domain ontology capturing the underlying data sources. A domain ontology can be additionally enriched with the business level vocabulary, to enable non-expert users to express their analytical needs. Notice for example a graphical representation of an ontology capturing the TPC-H[1] data sources in top-left part of Figure 2. Apart from manually defining requirements from scratch, *Requirements Elicitor* also offers assistance to end-users' data exploration tasks by analyzing the relationships in the domain ontology, and automatically suggesting potentially interesting analytical perspectives. For example, a user may choose the focus of an anal-

[1]`http://www.tpc.org/tpch/`



```
<MDschema>
 <facts>
  <fact>
   <name>fact_table_revenue</name>
...
```

```
<MDschema>
 <facts>
  <fact>
   <name>fact_table_netprofit</name>
...
```

```
<design>
 <metadata>...</metadata>
 <edges>  <edge>
  <from>DATASTORE_Partsupp</from>
...
```

```
<design>
 <metadata>...</metadata>
 <edges>  <edge>
  <from>DATASTORE_Nation</from>
...
```

```
<MDschema>
 <facts>
  <fact>
   <name>fact_table_revenue</name>
...
<MDschema>
 <facts>
  <fact>
   <name>fact_table_netprofit</name>
...
```

```
<design>
 <metadata>...</metadata>
 <edges>  <edge>
  <from>DATASTORE_Partsupp</from>
  <to>EXTRACTION_Partsupp</to>
  <enabled>Y</enabled>
 </edge> ...  </edges>
 <nodes>  <node>
 <name>DATASTORE_Partsupp</name>
 <type>Datastore</type>
...
```

```
CREATE DATABASE demo;
CREATE TABLE fact_table_revenue (
      Partsupp_PartsuppID BIGINT... ,
      Orders_OrdersID BIGINT ...,
      revenue double precision ,
PRIMARY KEY( Partsupp_PartsuppID,
          Orders_OrdersID )

); ...
CREATE TABLE fact_table_netprofit (
      ...
); ...
```

```
<transformation>
 <connection>
  ... <database>demo</database>...
<order>
 <hop>
 <from>DATASTORE_Partsupp</from>
 <to>EXTRACTION_Partsupp</to>
 <enabled>Y</enabled>
  </hop> ...
<step>
 <name>DATASTORE_Partsupp</name>
 <type>TableInput</type> ...
```

Figure 3: Design integration & deployment example

ysis (e.g., `Lineitem`), while the system then automatically suggests useful dimensions (e.g., `Supplier`, `Nation`, `Part`). The user can further accept or discard the suggestions and supply her information requirement.

2.2 Requirements Interpreter

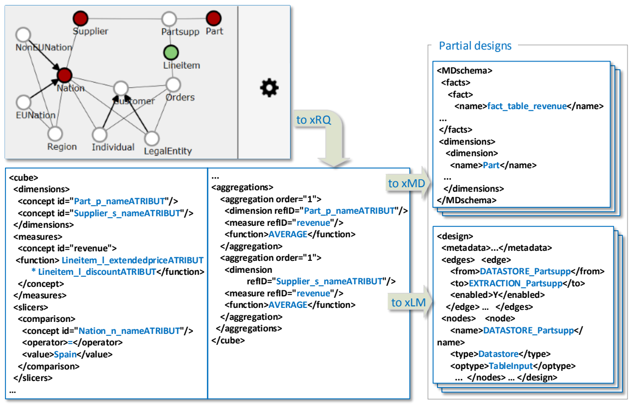
Each information requirement defined by a user, is then translated by the *Requirements Interpreter* to a *partial* DW design. In particular, *Requirements Interpreter* maps an input information requirement to underlying data sources (i.e., by means of a domain ontology that captures them and corresponding source schema mappings; see Section 2.5), and semi-automatically generates MD schema and ETL process designs that satisfy such requirement. For more details and a discussion on correctness we refer the reader to [11].

In addition, *Quarry* allows plugging in other external design tools, with the assumption that the provided *partial designs* are sound (i.e., meet MD integrity constraints) and that they satisfy an end-user requirement. To enable such cross-platform interoperability, *Quarry* provides logical, platform-independent representations (see Section 2.5). Generated designs are stored to the *Communication & Metadata layer* using corresponding formats and related to the information requirements they satisfy.

2.3 Design Integrator

Starting from each information requirement, translated to corresponding *partial* MD schema and ETL process designs, *Quarry* takes care of incrementally consolidating these designs and generating *unified design solutions* satisfying all current requirements (see Figure 3).

MD Schema Integrator. This module semi-automatically integrates *partial* MD schemas. *MD Schema Integrator*, comprises four stages, namely *matching facts*, *matching dimensions*, *complementing the MD schema design*, and *integration*. The first three stages gradually match different MD concepts and explore new DW design alternatives. The last stage considers these matchings and end-user's feedback to generate the final MD schema that accommodates new information requirements. To boost the integration of new information requirements spanning diverse data sources into the final MD schema design, we capture the semantics (e.g., concepts, properties) of the available data sources in terms of a domain ontology and corresponding source schema mappings (see Section 2.5). *MD Schema Integrator* automatically guarantees MD-compliant results and produces the optimal solution by applying cost models that capture different quality factors (e.g., structural design complexity).



```
<cube>
 <dimensions>
  <concept id="Part_p_nameATRIBUT"/>
  <concept id="Supplier_s_nameATRIBUT"/>
 </dimensions>
 <measures>
  <concept id="revenue">
 <function> Lineitem_l_extendedpriceATRIBUT
     * Lineitem_l_discountATRIBUT</function>
  </concept>
 </measures>
 <slicers>
  <comparison>
   <concept id="Nation_n_nameATRIBUT"/>
   <operator>=</operator>
   <value>Spain</value>
  </comparison>
 </slicers>
...
```

```
...
<aggregations>
  <aggregation order="1">
   <dimension refID="Part_p_nameATRIBUT"/>
   <measure refID="revenue"/>
   <function>AVERAGE</function>
  </aggregation>
  <aggregation order="1">
   <dimension
        refID="Supplier_s_nameATRIBUT"/>
   <measure refID="revenue"/>
   <function>AVERAGE</function>
  </aggregation>
 </aggregations>
</cube>
```

```
<MDschema>
 <facts>
  <fact>
   <name>fact_table_revenue</name>
 ...
 </facts>
 <dimensions>
  <dimension>
   <name>Part</name>
  ...
  </dimensions>
</MDschema>
```

```
<design>
 <metadata>...</metadata>
 <edges>  <edge>
  <from>DATASTORE_Partsupp</from>
  <to>EXTRACTION_Partsupp</to>
  <enabled>Y</enabled>
 </edge> ...  </edges>
 <nodes>  <node>
  <name>DATASTORE_Partsupp</
name>
  <type>Datastore</type>
  <optype>TableInput</optype>
   ...  </nodes> ... </design>
```

Figure 4: Example design process

ETL Process Integrator. This module processes *partial* ETL designs and incrementally consolidates them into a *unified* ETL design. *ETL Process Integrator*, for each new requirement maximizes the reuse by looking for the largest overlapping of data and operations in the existing ETL process. To boost the reuse of the existing data flow elements when answering new information requirements, *ETL Process Integrator* aligns the order of ETL operations by applying generic equivalence rules. *ETL Process Integrator* also accounts for the cost of produced ETL flows when integrating information requirements, by applying configurable cost models that may consider different quality factors of an ETL process (e.g., overall execution time).

More details, as well as the underlying algorithms of *MD Schema Integrator* can be found in [6] and of *ETL Process Integrator* in [5].

2.4 Design Deployer

Finally, *Quarry* supports the deployment of the unified design solutions over the supported storage repositories and execution platforms (see example in Figure 3). By using platform-independent representations of a DW design (see Section 2.5), *Quarry* is extensible in that it can link to a variety of execution platforms. At the same time, the validated DW designs are available for additional tunings by an expert user (e.g., indexes, materialization level).

2.5 Communication & Metadata Layer

To enable communication inside *Quarry*, the *Communication & Metadata layer* uses logical (XML-based) formats for representing elements that are exchanged among the components. Information requirements are represented in the form of analytical queries using a format called *xRQ*[2] (see bottom-left snippet in Figure 4). An MD schema is represented using the *xMD* format[3] (see top-right snippet in Figure 4), and an ETL process design using the *xLM* format [12] (see bottom-right snippet in Figure 4). Moreover, the *Communication & Metadata layer* offers plug-in capabilities for adding import and export parsers, for supporting various external notations (e.g., SQL, Apache PigLatin, ETL Metadata; see more details in [7]).

Besides providing the communication among different components of the system, the *Communication & Metadata layer*

[2]*xRQ*'s DTD at: `www.essi.upc.edu/~petar/xrq.html`
[3]*xMD*'s DTD at: `www.essi.upc.edu/~petar/xmd.html`

also serves as a repository for the metadata that are produced and used during the DW design lifecycle. The metadata used to boost the semantic-aware integration of DW designs inside the *Quarry* platform, are *domain ontologies* capturing the semantics of underlying data sources, and *source schema mappings* that define the mappings of the ontological concepts in terms of underlying data sources.

2.6 Implementation details

Quarry has been developed at UPC, BarcelonaTech in the last three years, using a service-oriented architecture.

On the client side, *Quarry* provides a web-based component for assisting end-users during the DW lifecycle (i.e., *Requirements Elicitor*). This component is implemented in *JavaScript*, using the specialized *D3* library for visualizing domain ontologies in form of graphs. The rest of modules (i.e., *Requirements Interpreter*, *MD Schema Integrator*, and *ETL Process Integrator*) are deployed on *Apache Tomcat 7.0.34*, with their functionalities offered via HTTP-based RESTful APIs. Such architecture provides the extensibility to *Quarry* for easily plugging and offering new components in the future (e.g., design self-tuning). Currently, all module components are implemented in *Java 1.7*, whilst new modules can internally use different technologies. For generating internal XML formats (i.e., *xRQ*, *xMD*, *xLM*) we created a set of *Apache Velocity 1.7* templates, while for their parsing we rely on the *Java SAX* parser. For representing domain ontology inside *Quarry*, we used *Web Ontology Language* (OWL), and for internally handling the ontology objects inside Java, we used the *Apache Jena* libraries. Lastly, the *Communication & Metadata layer*, which implements communication protocols among different components in *Quarry*, uses a *MongoDB* instance as a storage repository, and a generic XML-JSON-XML parser for reading from and writing to the repository.

3. DEMONSTRATION

In the on-site demonstration, we will present the functionality of *Quarry*, using our end-to-end system for assisting users in managing the DW design lifecycle (see Figure 1). We will use different examples of synthetic and real-world domains, covering a variety of underlying data sources, and a set of representative information requirements from these domains depicting typical scenarios of the DW design lifecycle. Demo participants will be especially encouraged to provide example analytical needs using *Requirements Elicitor*, and play the role of *Quarry*'s end-users. The following scenarios will be covered by our on-site demonstration.

DW design. Business users are not expected to have deep knowledge of the underlying data sources, thus they may choose to pose their information requirements using the domain vocabulary. To this end, business users may use the graphical component of *Quarry* (i.e., *Requirements Elicitor*), and its graphical representation of a domain ontology. This scenario shows how *Quarry* supports non-expert users in the early phases of the DW design lifecycle, to express their analytical needs (i.e., through assisted data exploration of *Requirements Elicitor*), and to easily obtain the initial DW design solutions.

Accommodating a DW design to changes. Due to possible changes in a business environment, a new information requirement could be posed or existing requirements might be changed or even removed from the analysis. Designers thus must reconsider the complete DW design to take into account the incurred changes. This scenario demonstrates how *Quarry* efficiently accommodates these changes and integrate them by producing an optimal DW design solution. We will consider *structural design complexity* as an example quality factor for output MD schemata, and *overall execution time* for ETL processes. The participants will see the benefits of integrated DW design solutions (e.g., reduced overall execution time for integrated ETL processes, executed in Pentaho PDI).

Design deployment. Finally, after the involved parties agree upon the provided solution, the chosen design is deployed on the available execution platforms. In this scenario, we will show how *Quarry* facilitates this part of the design lifecycle and generates corresponding executables for the chosen platforms. We use PostgreSQL for deploying our MD schema solutions, while for running the corresponding ETL flows, we use Pentaho PDI.

4. REFERENCES

[1] Z. E. Akkaoui, E. Zimányi, J.-N. Mazón, and J. Trujillo. A BPMN-Based Design and Maintenance Framework for ETL Processes. *IJDWM*, 9(3):46–72, 2013.

[2] L. Bellatreche, S. Khouri, and N. Berkani. Semantic Data Warehouse Design: From ETL to Deployment à la Carte. In *DASFAA (2)*, pages 64–83, 2013.

[3] S. Dessloch, M. A. Hernández, R. Wisnesky, A. Radwan, and J. Zhou. Orchid: Integrating Schema Mapping and ETL. In *ICDE*, pages 1307–1316, 2008.

[4] R. Fagin, L. M. Haas, M. A. Hernández, R. J. Miller, L. Popa, and Y. Velegrakis. Clio: Schema Mapping Creation and Data Exchange. In *Conceptual Modeling: Foundations and Applications*, pages 198–236. Springer, 2009.

[5] P. Jovanovic, O. Romero, A. Simitsis, and A. Abelló. Integrating ETL processes from information requirements. In *DaWaK*, pages 65–80, 2012.

[6] P. Jovanovic, O. Romero, A. Simitsis, A. Abelló, and D. Mayorova. A requirement-driven approach to the design and evolution of data warehouses. *Inf. Syst.*, 44:94–119, 2014.

[7] P. Jovanovic, A. Simitsis, and K. Wilkinson. Engine independence for logical analytic flows. In *ICDE*, pages 1060–1071, 2014.

[8] R. Kimball, L. Reeves, W. Thornthwaite, and M. Ross. *The Data Warehouse Lifecycle Toolkit.* J. Wiley & Sons, 1998.

[9] J.-N. Mazón, J. Lechtenbörger, and J. Trujillo. A survey on summarizability issues in multidimensional modeling. *Data Knowl. Eng.*, 68(12):1452–1469, 2009.

[10] C. Phipps and K. C. Davis. Automating data warehouse conceptual schema design and evaluation. In *DMDW*, volume 58 of *CEUR Workshop Proceedings*, pages 23–32, 2002.

[11] O. Romero, A. Simitsis, and A. Abelló. GEM: Requirement-Driven Generation of ETL and Multidimensional Conceptual Designs. In *DaWaK*, pages 80–95, 2011.

[12] A. Simitsis and K. Wilkinson. The specification for xLM: an encoding for analytic flows, HP Technical Report, 2015.

QaRS: A User-Friendly Graphical Tool for Semantic Query Design and Relaxation

Géraud Fokou
LIAS/ISAE-ENSMA
University of Poitiers
Futuroscope Cedex - France
geraud.fokou@ensma.fr

Stéphane Jean
LIAS/ISAE-ENSMA
University of Poitiers
Futuroscope Cedex - France
stephane.jean@ensma.fr

Allel Hadjali
LIAS/ISAE-ENSMA
University of Poitiers
Futuroscope Cedex - France
allel.hadjali@ensma.fr

Mickaël Baron
LIAS/ISAE-ENSMA
University of Poitiers
Futuroscope Cedex - France
mickael.baron@ensma.fr

ABSTRACT

This paper presents a *Query-and-Relax System* (`QaRS`) designed to facilitate the exploitation of large knowledge bases. `QaRS` proposes a graphical interface to construct a `SPARQL` query and use different cooperative answering techniques. The proposed cooperative techniques help users in finding alternative answers when their queries fail or do not return the expected number of answers. The present demonstration includes three main relaxation strategies: (1)- *automatic* where the system automatically relaxes the query based on similarity measures, (2)- *manual* where the user can specify the conditions that can or cannot be relaxed as well as the tolerance values and (3)- *interactive* where `QaRS` computes the causes of the query failure as a set of Minimal Failing Subqueries (`MFSs`) and then the user chooses the relaxation operators according to these `MFSs`.

General Terms

Algorithms, Design, Experimentation

Keywords

SPARQL, Relaxation, Minimal Failing Subquery, Similarity

1. INTRODUCTION

In recent years, several large Knowledge Bases (`KBs`) have been created such as YAGO [5] or Knowledge Vault [1] which contain millions of entities and facts about them. Such information is usually stored in `RDF` format and queried with the `SPARQL` language. Large `KBs` are difficult to use as (1)- their schema (often called ontology) and its underlying semantics are rarely understood by end users and (2)- `RDF` can be used to represent data ranging from unstructured to structured data leading to more or less sparse data [2]. A common issue encountered by users is the problem of failing queries, i.e., query results are empty or do not contain the number of expected answers.

As an example, let us consider the ontology inspired by the `LUBM` Benchmark depicted in Figure 1. If a user wants to find the professors who are assisted by one of her/his *phD* student in an *UnderGraduateCourse*, (s)he may write the query:

```
 SELECT ?X ?Y ?Z
WHERE { ?Z ub:teacherOf ?Y.
   ?Y rdf:type ub:UnderGraduateCourse.
   ?X ub:teachingAssistantOf ?Y.
   ?Z ub:advisorOf ?X.
   ?X rdf:type ub:AssistantProfessor. }
```

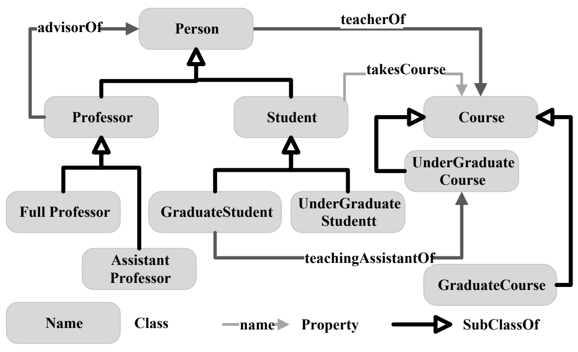


Figure 1: Ontology Example

In this query, the user makes the false assumption that a teaching assistant of a course is an *AssistantProfessor* (instead of a *GraduateStudent*). With a deeper knowledge of the ontology, the user could have known that the *teachingAssistantOf* property has the *GraduateStudent* class as domain and thus his/her query can not return any result.

Moreover, even if the query was written without any misconception, the query could still have failed if it is too restrictive or if the target `KB` is incomplete. To solve these problems, several works have proposed relaxation techniques for SPARQL queries (e.g., [3, 7]). But none of them proposes a simple and intuitive graphical system to build a SPARQL query and relax it with or without the help of the user. Conversely, several works have proposed graphical systems to query large knowledge bases (e.g., [8, 9]). But they do not support any cooperative query answering technique needed to return alternative answers to failing queries. The QaRS system described in this paper aims at filling this gap. First, the functionalities of this system and its architecture are discussed. Then, the demonstration scenarios that we intend to show the audience are presented.

2. QARS'S FUNCTIONALITIES

The system we propose has three main functionalities, (i) a visual assistant for designing consistent queries, (ii) a visual help for queries relaxation and (iii) an explanation of both queries's failure and relaxation process.

2.1 Graphical query design

The ontology browser panel displays the ontology as a graph (see Figure 2). As an ontology can be large, a search box with auto completion feature is available for finding classes and properties that will be used in the query. When a class or property is selected, the graph is centered around this concept to see all the related concepts.

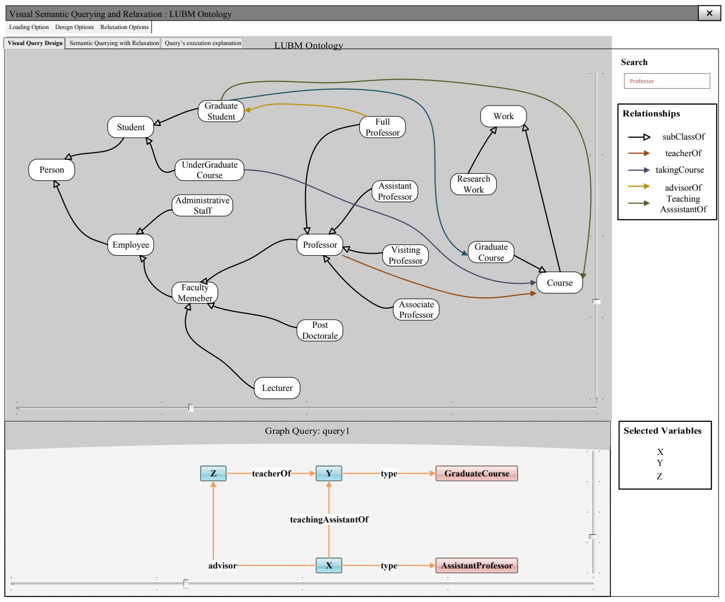


Figure 2: QaRS Query Design Panel

The visual construction of a query is composed of three main steps:
1) dragging and dropping classes and/or properties from the ontology browser into the query panel. Dropping a class C in this panel creates a graph corresponding to the triple ($?v_i$ *rdf:type* C) where v_i is a variable which has not been previously used in the query. In the same way, dropping a property P creates a graph showing the triple ($?v_i$ P $?v_j$).
2) Linking the triple patterns defined in the previous step by identifying the variables that they share. This action is done by dragging a variable and dropping it into an other variable. It indicates that the two variables are the same. In this step, QaRS checks whether the query can return a result by testing its consistency w.r.t. the domain and range of the properties. In our running example, since we know that X is a teaching assistant of the course Y, according to the ontology, one can deduce that X can not be an $AssistantProfessor$. So the last triple of this query leads to an inconsistency. As another example, the query Q:

```
SELECT ?X ?Y WHERE {
   ?X rdf:type ub:GraduateStudent.
   ?X ub:teachingAssistantOf ?Y.
   ?Y rdf:type ub:GraduateCourse. }
```

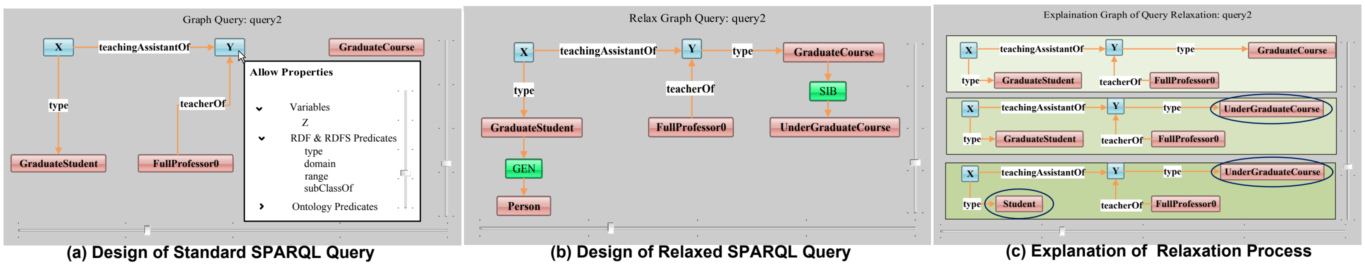
is consistent when it has only the first two triple patterns as depicted in Figure 3-(a) (*there are graduate students who are teaching assistants*). But when the last triple pattern is added, the query becomes inconsistent, i.e. it can not return any answer (*graduate students can not be teaching assistants of a graduate course*). In this case, QaRS alerts the users.

3) adding FILTER, OPTIONAL and/or UNION operators by selecting the components of the query on which they must be applied (a variable, a triple or a set of triples), right clicking on them and choosing the corresponding operators. Each one of these operators is graphically identified in the query by a specific color or component. The SPARQL query corresponding to the graphical query can also be displayed and the user can interact both with the textual or graphical query to edit it. Modifying the graphical query automatically changes the textual query and vis-versa.

2.2 Query relaxation strategies

Automatic relaxation. When executing the query designed in the previous step, the user can specify the minimum number, say k, of expected answers. If the query result does not have k answers, QaRS considers it as a failing query and automatically relaxes it. Basically, this automatic relaxation process consists in computing a set of possible relaxed queries (see further) from the initial query, ordering them according to their similarities with the initial query and executing them following this order until the number of expected results is reached. If the result of a relaxed query is large, the answers are ordered according to their satisfaction degrees w.r.t. the initial query and then the user is provided with the $top - k$ answers.

Manual relaxation. Users may have some constraints on the parts of the query that can be relaxed as well as on the tolerance values that are acceptable. They can manually specify these constraints in the design query panel. Three kind of constraints can be graphically defined by users. (i) Triple patterns that must not be relaxed. The user simply selects a subset of the query graph that must not be relaxed, right clicks on it and selects the corresponding option. (ii) Allowed classes (resp. properties) in the hierarchy of a class (resp. property) that can be used to relax the query. The user selects a class or property, right clicks on it and selects the relaxation option. The hierarchy of the class or property is then displayed and the user can select the classes or properties that can be used in the relaxation process (see Figure 3-(b)). In this step, QaRS checks that the selected classes or properties lead to a consistent relaxed query. (iii) Allowed values to relax filters. In a similar way as above, the user selects a filter that can be relaxed, right clicks on it and



(a) Design of Standard SPARQL Query (b) Design of Relaxed SPARQL Query (c) Explanation of Relaxation Process

Figure 3: User interface of the System

selects the relaxation option. According to the datatype of the filter, a panel allows the user to define the tolerance values. Thanks to methods borrowed from fuzzy logic theory, one can obtain satisfaction degrees of the relaxed value of the filter. Finally, the user specifies the minimum number of expected results and executes the query. If the query fails, QaRS triggers the relaxation process while respecting his/her constraints (see Figure 3-(c)).

Interactive relaxation. In the previous scenario, the user does not know the causes of the failure of his/her query. QaRS can provide him/here with explanation. This explanation consists in displaying a set of Minimal Failing Subqueries (MFSs) [4] of the query. For the query Q (section 2.1), the cause of its failure is the subquery below.

```
 SELECT ?X ?Y
WHERE { ?X ub:teachingAssistantOf ?Y.
          ?Y rdf:type ub:GraduateCourse. }
```

Each MFS (i) is a failing subquery of the initial query and (ii) does not include a failing subquery. Thus, if the MFSs of a query are not relaxed, the initial query will never return non-empty answers. In this scenario, the relaxation process is a two-step procedure: (i) QaRS displays the set of MFSs of the query; (ii) the user can automatically or manually relax each MFS like in the previous scenarios. By default, QaRS proposes to make optional the triple patterns of the MFSs.

2.3 Explanation and customization

Similarity is a key notion in QaRS. It is leveraged by the system, on the one hand, for measuring the similarity between the initial query and the relaxed ones and, on the other hand, for computing the satisfaction degrees of alternative answers returned w.r.t the initial query. The latter point allows to provide user with discriminated set of answers and then (s)he can select the *top-k* answers (where k is the minimum number of expected answers). As for query similarity, it helps users to rank-order the relaxed queries and choose an appropriate set of queries to be executed to obtain k answers. These executed queries can be seen as an explanation for the user to (progressively) reach the desired answers. As different similarity measures can be used to compute the similarity between two classes (or properties) of an ontology, relaxation performed by QaRS can be customized by selecting measures that fit best the user's needs.

3. SYSTEM ARCHITECTURE

The architecture of QaRS is illustrated in Figure 4. It comprises two main parts. The first part includes two components: *Graphical Design Of* SPARQL *Query* (GDSQ) and SPARQL *Query Analyzer* (SQLA). While the second part, which is related to the core of query relaxation, is composed of four modules: *Relaxation Operator Interpretor*, *Automatic Query Relaxation*, MFS *Engine for* SPARQL *Query* and *Ranking Alternative Answers Engine*. The module *Extended* SPARQL *Query Engine* is an extension of standard SPARQL query engine which allows us to launch the relaxation process when the query at hand fails. This module also makes easy the integration of QaRS in any triplestore environment. Now, we provide details about each component of QaRS.

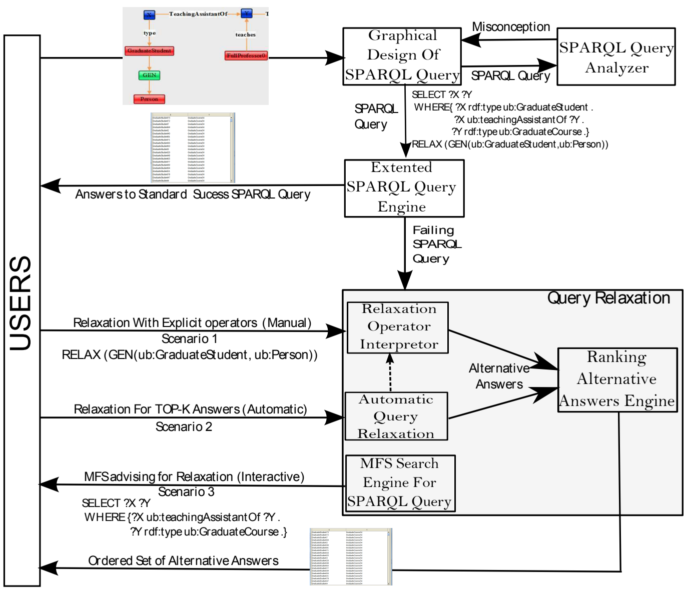

USERS
TeachingAssistantOf
type
teaches
GraduateStudent
FullProfessor0
GEN
Person
Graphical Design Of SPARQL Query
Misconception
SPARQL Query
SPARQL Query Analyzer
SPARQL Query
SELECT ?X ?Y
WHERE{ ?X rdf:type ub:GraduateStudent .
?X ub:teachingAssistantOf ?Y .
?Y rdf:type ub:GraduateCourse .}
RELAX (GEN(ub:GraduateStudent,ub:Person))
Extented SPARQL Query Engine
Answers to Standard Sucess SPARQL Query
Failing SPARQL Query
Query Relaxation
Relaxation Operator Interpretor
Automatic Query Relaxation
MFS Search Engine For SPARQL Query
Ranking Alternative Answers Engine
Alternative Answers
Relaxation With Explicit operators (Manual)
Scenario 1
RELAX (GEN(ub:GraduateStudent, ub:Person))
Relaxation For TOP-K Answers (Automatic)
Scenario 2
MFS advising for Relaxation (Interactive)
Scenario 3
SELECT ?X ?Y
WHERE {?X ub:teachingAssistantOf ?Y .
?Y rdf:type ub:GraduateCourse .}
Ordered Set of Alternative Answers

Figure 4: Architecture of QaRS

GDSQ and SQA modules

GDSQ offers a user-friendly interface to assist users in the design and building of their SPARQL queries in a graphical and intuitive way. As for SQA module, which is an online analyzer, it checks on-the-fly the syntax and the consistency of the designed query. It also proposes auto completion and suggests concepts for designing the query.

Relaxation interpretor

This module interprets each of the three relaxation operators studied in [3] and generates the corresponding set of relaxed SPARQL queries. *GEN* operator takes as input a concept C_i and a super concept C_f. The system generates SPARQL queries with C_i replaced by the classes in the path from C_i

to C_f in the ontology. As for SIB operator, the system generates SPARQL queries where a concept C_0 is replaced by its sibling classes $C_1, C_2, ..., C_m$. In the case of $PRED$ operator, it incrementally relaxes filters involving simple data types (numeric, string, etc). This module is launched when the clause $RELAX$ is used in the designed SPARQL query.

Automatic query relaxation

This module is called when users want to have the *top-k* answers without setting the relaxation operators to use. QaRS generates all the relaxed queries using the three previous operators and their combinations, in the spirit of the approach proposed by [6]. A rank-ordering of these queries is established according to their similarity w.r.t. the failing query. The following similarity measure between classes is used [3]:
$Sim(C_i, C_f) = \dfrac{IC(msca(C_i, C_f))}{IC(C_i) + IC(C_f) - IC(msca(C_i, C_f))}$
Then, the relaxed queries are executed from the most similar to the least similar and the answers obtained are sent to the *Ranking alternative answers engine*.

Ranking alternative answers engine

QaRS provides the user with a set of alternative answers in a discriminated way. Each answer h_i is associated with a satisfaction score computed as follows [3]:
$SatQ(h_i) = \min(Sim(Q', Q), SatQ'(h_i))$
where $Sim(Q, Q')$ stands for the similarity measure between the initial query Q and its relaxed form Q' and $SatQ'(h_i)$ for a satisfaction degree of h_i w.r.t. Q'. This latter degree is obtained thanks to the formula [3]:
$SatQ'(h_i) = min(\max\limits_{t \in type(h_i)} Sim(t, c'), \mu_p(h_i.propRelax))$
where c' is the relaxed class which gives the answer h_i and μ_p is the membership function of the (fuzzy) property $propRelax$.

MFS search engine

The MFS search engine is a module which identifies the causes of query failure. To do so, a set of MFSs of the failing query are computed. MFSs provide user with a clear explanation on the empty answer problem. First, we transform the target SPARQL query into a set of triple patterns to form a conjunctive query. Next, an MFS of the conjunctive query is computed. To find the other MFSs, a set of significant subqueries (SSQs) is calculated. Each SSQ is characterized by three properties: (i) it does not contain the MFSs found; (ii) it is not included in those MFSs and, (iii) it does not include any other SSQ. The above two step-procedure is executed recursively on each element of the set of SSQs. All MFSs produced by this procedure are shown to the user as an explanation about his/her query failure.

4. DEMO SCENARIOS

We run two scenarios on LUBM ontology data. The first scenario aims at relaxing a failing query Q manually. The second one shows the interest of the MFSs as explanation of the query failure and their use for an efficient relaxation. To run the above scenarios, we use Jena triplestore to load the generated LUBM′s data.

Scenario 1

The user wants to find all *"the graduate students who are teaching assistants of a graduate course"*. The SPARQL query for this request is given in section 2.1. To obtain non empty answers, the user can ask a generalization (resp. substitution) of *GraduateStudent* (resp. *GraduateCourse*) concept to *Person* (resp. with *UnderGraduateCourse*) concept. This can be done graphically as shown in figure 3-(b). The relaxation operators proposed by QaRS depend of the query's concept to relax and ontology. As it can be seen in the ontology of Figure 1, the relaxed query may result in non empty answers since *GraduateStudent* may be teaching assistants of $UnderGraduateCourse$ which is a sibling $Class$ of *GraduateCourse*. It is worthing to note that this kind of relaxation does not always guarantee the success of the relaxation process. It is the case for the generalization of *GraduateStudent* to *Person* (where there is none subclass of *Person* with *teachingAssistantOf* as property, except the subclass *GraduateStudent*).

Scenario 2

To avoid the main flaw of the above scenario, we first identify the causes (i.e., MFSs) of query failure, then we apply appropriate relaxations on the triple patterns involved in the MFSs of the query. For our running example of section 2.1, which contains one MFS (see section 2.2), QaRS identifies it and shows this MFS graphically to the user. Then, the system suggests appropriate operators for relaxing the MFS. In our case, GEN or SIB operator will be proposed to relax the triple (?Y *rdf*:*type ub*:*GraduateCourse*) included in the MFS. By this way, alternative answers that fit best the user's needs are returned by the system.

5. REFERENCES

[1] X. Dong, E. Gabrilovich, G. Heitz, W. Horn, N. Lao, K. Murphy, T. Strohmann, S. Sun, and W. Zhang. Knowledge Vault: A Web-scale Approach to Probabilistic Knowledge Fusion. In *ACM SIGKDD, (KDD '14)*, pages 601–610, 2014.

[2] S. Duan, A. Kementsietsidis, K. Srinivas, and O. Udrea. Apples and Oranges: A Comparison of RDF Benchmarks and Real RDF Datasets. In *SIGMOD '11*, pages 145–156, 2011.

[3] G. Fokou, S. Jean, and A. Hadjali. Endowing Semantic Query Languages with Advanced Relaxation Capabilities. In *ISMIS'14*, pages 512–517, 2014.

[4] P. Godfrey. Minimization in Cooperative Response to Failing Database Queries. *International Journal of Cooperative Information Systems*, 6(2):95–149, 1997.

[5] J. Hoffart, F. M. Suchanek, K. Berberich, and G. Weikum. YAGO2: A spatially and temporally enhanced knowledge base from wikipedia. *Artificial Intelligence*, 194:28–61, 2013.

[6] H. Huang, C. Liu, and X. Zhou. Approximating Query Answering on RDF Databases. *World Wide Web*, 15(1):89–114, 2012.

[7] C. A. Hurtado, A. Poulovassilis, and P. T. Wood. Query Relaxation in RDF. *Journal on data semantics X*, pages 31–61, 2008.

[8] N. Jayaram, M. Gupta, A. Khan, C. Li, X. Yan, and R. Elmasri. GQBE: Querying knowledge graphs by example entity tuples. In *IEEE ICDE'14*, pages 1250–1253, 2014.

[9] A. Russell and P. R. Smart. NITELIGHT: A Graphical Editor for SPARQL Queries. In *Proceedings of the Poster and Demonstration Session at ISWC'08*, 2008.

Using Object-Awareness to Optimize Join Processing in the SAP HANA Aggregate Cache

Stephan Müller
Hasso Plattner Institute
Potsdam, Germany
stephan.mueller@hpi.de

Anisoara Nica
SAP SE
Waterloo, Canada
anisoara.nica@sap.com

Lars Butzmann
Hasso Plattner Institute
Potsdam, Germany
lars.butzmann@hpi.de

Stefan Klauck
Hasso Plattner Institute
Potsdam, Germany
stefan.klauck@hpi.de

Hasso Plattner
Hasso Plattner Institute
Potsdam, Germany
hasso.plattner@hpi.de

ABSTRACT

The introduction of columnar in-memory databases, along with hardware evolution, has made the execution of transactional and analytical workloads on a single system both feasible and viable. Yet, doing analytics directly on the transactional data introduces an increasing amount of resource-intensive aggregate queries which can slow down the overall system performance in a multi-user environment. To increase the scalability of a system in the presence of multiple such queries, we propose an aggregate cache in the general delta-main architecture that provides an efficient means to handle costly aggregate queries by applying incremental materialized view maintenance and query compensation techniques. Handling aggregate queries based on joins of multiple tables however is still a challenge as query compensation can be very expensive in the delta-main architecture of columnar in-memory databases. Our analysis of enterprise applications has revealed several data schema and workload patterns that can be leveraged for addressing performance of query processing using the aggregate cache. We contribute by presenting an approach to transport the application object semantics into the database system, becoming object-aware, and optimize the query processing using the aggregate cache by applying partition pruning and predicate pushdown in such general delta-main architecture. Our experimental validation using customer data and workloads confirms that this type of optimizations enables efficient usage of the aggregate cache for an even higher share of aggregate queries as one mean to scale the system.

1. INTRODUCTION

The separation of enterprise applications into online transactional processing (OLTP) and online analytical processing (OLAP) induces drawbacks including stale and redundant data, and inflexible analytics due to pre-calculated data cubes. A closer look reveals that this separation is mostly artificial as both systems have the same number of inserts – unless the OLAP system already abstracts from the base data – and a high share of analytical queries with costly aggregations. To deal with aggregate queries, both systems employ different approaches. While OLAP systems make extensive use of materialized views [29, 33], we see that the handling of aggregates in OLTP systems is often done within the application by maintaining predefined summary tables. This leads to an increased application complexity with risks for violating data consistency and to a limited throughput of insert and update queries as the related summary tables must be updated in the same transaction [14, 25].

With the ongoing trend of columnar in-memory databases (IMDBs) such as Hyrise [11], SAP HANA [9], and Hyper [16], this artificial separation is no longer necessary as they are capable of handling mixed workloads, with transactional and analytical queries, in a single system [24]. In columnar IMDBs, the storage is separated into a highly compressed, read-optimized *main* storage and a write-optimized *delta* storage, both implemented as columnar data stores. New records are inserted to the delta storage and periodically *merged* to the main storage [17]. Having a single IMDB for transactional and analytical workloads however imposes one central challenge: While modern hardware enables the execution of arbitrary complex computations in a short time by parallelization, this means that one query can saturate an arbitrary large machine [30]. Especially the execution of expensive aggregations that may be done by many hundreds of users in parallel is problematic and requires means to keep the system scalable.

Despite the aggregation capabilities of columnar IMDBs [24], access to tuples of a materialized aggregate – which we define as a materialization of a query which contains aggregate functions – is always faster than aggregating on the fly. However, the overhead of materialized view maintenance to ensure consistency for modified base data has to be considered and involves several challenges [12]. It turns out that the main-delta architecture is well-suited for the *aggregate cache*, a novel strategy of dynamically caching aggregate queries and applying incremental view maintenance techniques [21] for maintaining the cache and answering queries using the aggregate cache. In the general main-delta architecture,

10.5441/002/edbt.2015.57

only the delta storage is updated when data is modified, for example, when new records are inserted. In our design of the aggregate cache, the materialized aggregates are only defined on records from the main storage. Hence, the materialized aggregates do not have to be invalidated when new records are inserted, updated, or deleted in the delta storage. When a query result is computed using the aggregate cache, the final, consistent query result is delta-compensated, on the fly, by aggregating the newly inserted records of the delta storage and combining them with the previously cached aggregate of the main storage.

One challenge of the aggregate cache in the main-delta architecture is to achieve high performance for relevant classes of application queries which include aggregates based on joins of multiple tables. These queries require expensive delta-compensations based on subjoins of all permutations of delta and main partitions of the involved tables, excluding the already cached join of the main partitions. For a query joining two tables, three extra subjoins are required for delta-compensation, and a query joining three tables already requires seven extra subjoins. This may result in very little performance gains over not using the aggregate cache. However, after analyzing the characteristics of several enterprise applications, we identified schema design and workload patterns that can be leveraged to allow pruning certain subjoins and therefore optimize the overall performance of join queries using the aggregate cache.

In this paper, we make the following contributions:

- We introduce the aggregate cache, a materialized aggregate engine implemented in SAP HANA, leveraging the main-delta architecture of columnar IMDBs, and describe its current architecture (see Section 2.1).
- We discuss the query processing using the aggregate cache and performance challenges related to main and delta compensations which are metrics for admittance in the aggregate cache (see Sections 2.2 and 2.3).
- We identify a class of join queries which normally do not qualify to be admitted in the aggregate cache and analyze their performance issues when using the aggregate cache. We propose a novel solution for increasing the performance for this class of queries exploiting the main-delta architecture and the application object semantics. These techniques are implemented as a prototype which extends the aggregate cache for join queries. The main contributions here are:
  - We analyze enterprise applications which benefit the most from the dynamic aggregate cache and identify several schema design and workload patterns imposed by the object semantics of these applications (see Section 3).
  - We give a formal definition of the join pruning problem in the aggregate cache and define matching dependencies among relations based on join attributes and temporal relationships as one possible design for efficiently using the aggregate cache for join queries (see Section 4).
  - We discuss our implementation for transporting application object semantics into the database to become *object-aware* which allows join pruning techniques to be applied for queries using the aggregate cache (see Section 5).
- We use the CH-benCHmark [10], a benchmark based on TPC-H and TPC-C, and a benchmark using real customer workloads from a production Enterprise Resource Planning (ERP) system to show performance results for (1) aggregate cache maintenance strategies; (2) data update overhead for tables referenced in the aggregate cache; (3) query processing using aggregate cache with and without join pruning (see Section 6).

2. AGGREGATE CACHE

The aggregate cache leverages the concept of the main-delta architecture in SAP HANA [9]. Separating a table into a main and delta storage has one main benefit: it allows to have a read-optimized main storage for faster scans and a write-optimized delta storage for high insert throughput. All records in the delta storage are periodically propagated into the main storage in an operation called *delta-merge* [17]. The fact that new records are added to the main storage only during a merge operation is leveraged by the aggregate cache which is designed to cache only the results computed on the main storage. For a current query using the aggregate cache, the records from the delta storage are aggregated on-the-fly which compensates the corresponding cache entry to build the result set of the query, a process we refer to as *delta compensation*. In the general main-delta architecture, records are not updated in place. Instead, the updated record is inserted in the delta partition whereas the old record in the main (or delta) partition is *invalidated*. Other database implementations with a delta storage or differential buffer such as C-Store, Sybase IQ, MonetDB/X100, Hyrise, or memory-optimized tables in SQL Server handle updates very similarly to the mechanism implemented in SAP HANA. During the

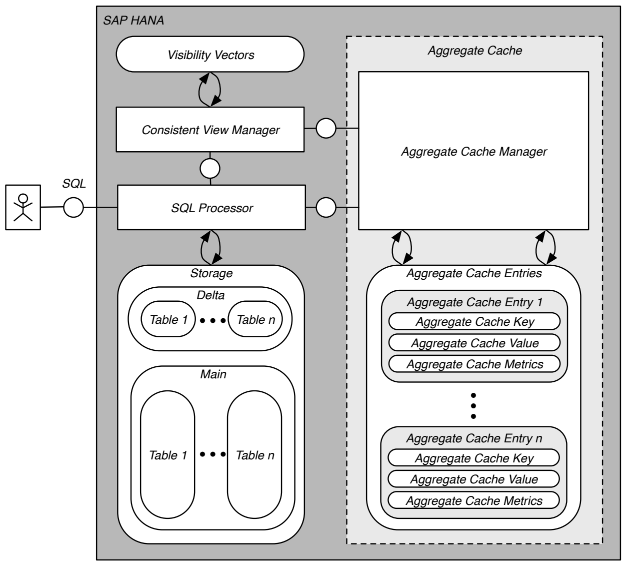


Figure 1: The architecture of the aggregate cache in SAP HANA.

next merge, all invalidated records can either be removed from the main storage or kept so that temporal query processing on historical data can be supported [15]. To handle invalidations in the main partition, we apply a *main compensation* process as described in Section 2.2.

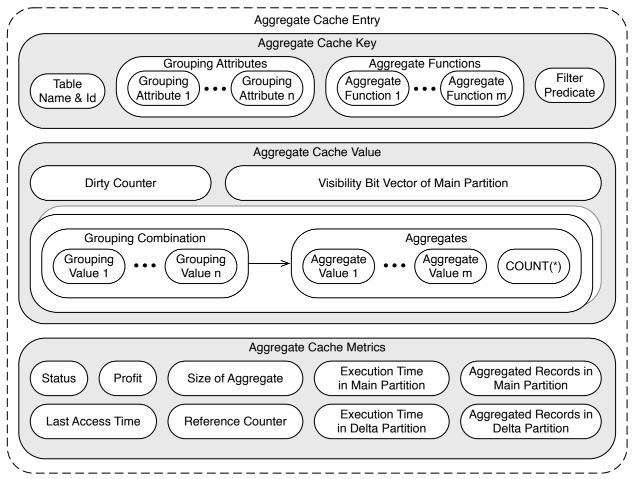


Figure 2: The structure of an aggregate cache entry, consisting of an aggregate cache key, an aggregate cache value, and aggregate cache metrics.

2.1 Architecture

As illustrated in Fig. 1, the aggregate cache is implemented inside the column store engine of SAP HANA [9]. The *aggregate cache manager* is the core component of the aggregate cache, managing aggregate cache entries.

An aggregate cache entry, depicted in Fig. 2, consists of a key, a value, visibility vectors, and profit metrics. The *aggregate cache key* is a unique identifier based on the query definition including the table name, table id, the grouping attributes, the aggregate functions, and the filter predicates of the related aggregate query. The *aggregate cache value*, the extent of the aggregate query, is a structure consisting of the grouping combinations and the corresponding aggregate functions: it contains the result set of the aggregate computed only on the main storage. The aggregate cache entry further contains dirty counters that indicates if records have been invalidated in the main partitions, and the visibility vectors of the main partitions at the time of last computation. The aggregate cache entry is first created during query processing (Fig. 3) and it is maintained during the delta-merge operations. *Aggregate cache metrics* are maintained for each entry including the aggregate's size, the number of aggregated records, execution times for delta and main compensations, maintenance times, and usage information. The metrics are required to calculate the profit of an aggregate cache entry to be used for dynamic cache admission, eviction, and maintenance decisions [20].

Query execution using the aggregate cache is shown in Fig. 3: the query executor delegates aggregate query blocks that qualify for the aggregate cache to the *aggregate cache manager*. The aggregate cache supports queries with *self-maintainable* aggregate functions [22] including SUM, COUNT, and AVG. When the aggregate cache matching process is not successful, the aggregate cache manager attempts to create a cache entry by executing the aggregate query on the main partitions with the global record visibility which is retrieved through the *consistent view manager*. If the aggregate is profitable enough for cache admission, the result is used to create an aggregate cache entry. In both cases, when aggregate entry is retrieved from the cache or it is just cached by the current transaction, the *main compensation* and the *delta compensation* must be applied.

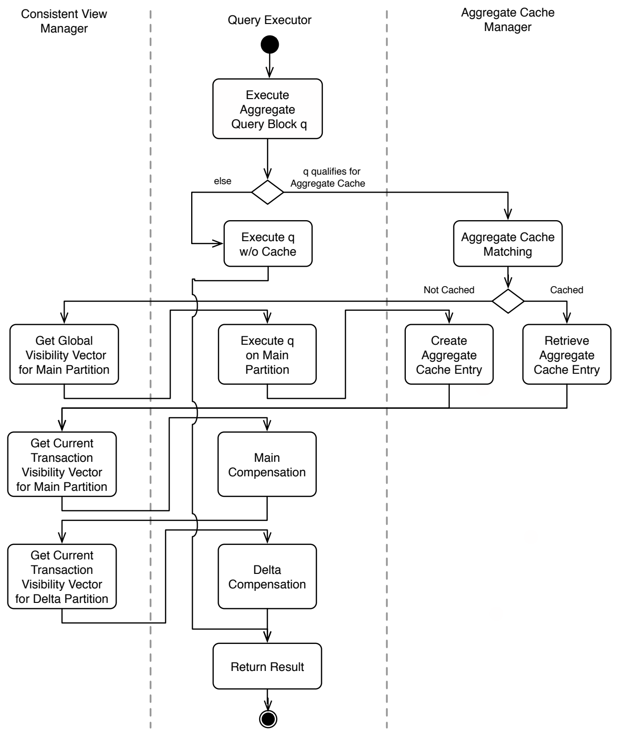

Consistent View Manager
Query Executor
Aggregate Cache Manager
Execute Aggregate Query Block q
else
q qualifies for Aggregate Cache
Execute q w/o Cache
Aggregate Cache Matching
Not Cached
Cached
Get Global Visibility Vector for Main Partition
Execute q on Main Partition
Create Aggregate Cache Entry
Retrieve Aggregate Cache Entry
Get Current Transaction Visibility Vector for Main Partition
Main Compensation
Get Current Transaction Visibility Vector for Delta Partition
Delta Compensation
Return Result

Figure 3: Query processing with the aggregate cache: creation and usage of aggregate cache entries including main and delta compensation during query execution.

2.2 Main Compensation

While updates and deletes of records in the delta storage are handled transparently and do not affect our caching algorithm, an aggregate cache entry can become inconsistent with respect to a record invalidation in the main storage, including deletes and updates of the current transaction. Instead of recalculating an aggregate cache entry with every record invalidation in the main storage, we employ an approach that uses bit vector comparison to efficiently detect invalidated records and apply them to aggregate cache entries in a process called *main compensation.* As illustrated in Fig. 3, we use the consistent view manager to retrieve current record visibilities during aggregate cache entry usage.

The record invalidation is handled through the consistent view manager (see Fig. 1) that creates a bit vector representing the visibility of records of a table for an incoming query based on its transaction token. When an aggregate query is cached, the current snapshot is captured using this visibility vector. When a query is executed using an aggregate cache entry, an efficient bit vector comparison of the current snapshot with the snapshot at the cache creation time is used, thereby detect invalidated records, and apply them for main compensation. The details of aggregate cache main compensation can be found in [19] and are omitted in this paper for simplification reasons.

2.3 Delta Compensation

As the last step in query execution (Fig. 3), any query using the aggregate cache must apply delta compensation operation which accounts for records in the delta storage visible to the current transaction. When the aggregate cache is based on multiple tables joins, the complexity of answering a query using the aggregate cache increases as the aggregate cache is computed on the main partitions only, and the query must be compensated with all subjoins on deltas and mains. As a result, the profit of caching an aggregate query based on many tables may be very low because their performance using the aggregate cache is not superior to not using it. The techniques proposed in this paper have the main goal of extending the class of aggregate queries which qualify to be admitted into the SAP HANA aggregate cache.

The classical aggregate query joining a header table H, an item table I, and a dimension table D (see Section 3) on the join conditions $H[A] = I[A]$ and $I[B] = D[B]$ is $Q(H, I, D) = H \bowtie_{H[A]=I[A]} I \bowtie_{I[B]=D[B]} D$. In main-delta architecture, each table X consists of at least two partitions $\mathbb{P}(X) = \{X_{main}, X_{delta}\}$ which adds complexity when the result of the query $Q(H, I, D)$ is computed as the join processing must consider all subjoin combinations among these partitions. Theoretically, the subjoins on delta and main partitions of the tables referenced in $Q(H, I, D)$ are as depicted in Equation 1 and Fig. 4. The subscript numbers in Equation 1 of the subjoins match the subjoin numbers in Fig. 4. Based on the size of the involved table components, the time to execute the subjoins varies. Typically, the ratio between the sizes of main and delta partitions is 100:1. In our example the subjoins #5 and #8 require the longest time, since they involve matching the join condition of the mains of two large tables.

$$\begin{aligned} &Q(H, I, D) = \\ &(H_{delta} \bowtie_{H[A]=I[A]} I_{delta} \bowtie_{I[B]=D[B]} D_{main})_1 \\ &\ldots \cup (H_{main} \bowtie_{H[A]=I[A]} I_{main} \bowtie_{I[B]=D[B]} D_{delta})_5 \\ &\ldots \cup (H_{main} \bowtie_{H[A]=I[A]} I_{main} \bowtie_{I[B]=D[B]} D_{main})_8 \end{aligned} \tag{1}$$

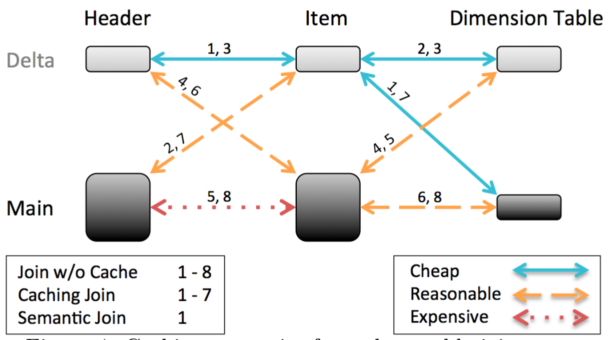


Figure 4: Caching strategies for a three table join query.

2.3.1 *Join without Aggregate Cache*

Join queries referencing partitioned tables are of the form $Q(R_1, \ldots, R_t) = R_1 \bowtie_{c_1(R_1,R_2)} \cdots \bowtie_{c_{t-1}(R_{t-1},R_t)} R_t$, where each table R_i has the partitioning $\mathbb{P}(R_i) = \{R_{i,1}, \ldots, R_{i,k_i}\}$, for all $i \in \{1, \ldots, t\}$. Without caching some of the subjoins, the database engine needs to compute all possible join combinations of the involved number of tables t and partitions $\mathbb{P}(R_i)$ to build a complete result set. The result of Q is a union of all $k_1 \times \ldots \times k_t$ subjoins i.e., $Q(R_1, \ldots, R_t) = \bigcup_{(j_1,j_2,\ldots,j_t) \in \mathbb{J}_{noCache}(Q)} R_{1,j_1} \bowtie_{c_1(R_1,R_2)} \cdots \bowtie_{c_{t-1}(R_{t-1},R_t)} R_{t,j_t}$, with $\mathbb{J}_{noCache}(Q) = \{1, \ldots, k_1\} \times \cdots \times \{1, \ldots, k_t\}$.

To evaluate $Q(H, I, D)$ from Equation 1, joining three tables with two partitions each, that adds up to a total of $2^3 = 8$ subjoins to be unified: $Q(H, I, D) = \bigcup_{(j_1,j_2,j_3) \in \mathbb{J}_{noCache}(3)} (H_{j_1} \bowtie_{H[A]=I[A]} I_{j_2} \bowtie_{I[B]=D[B]} D_{j_3})$, where $\mathbb{J}_{noCache}(3) = \{delta, main\} \times \{delta, main\} \times \{delta, main\}$.

2.3.2 *Join with Aggregate Cache*

When using the aggregate cache, the result set from joining all main partitions is already cached (i.e., $R_{1,main} \bowtie \cdots \bowtie R_{t,main}$) and the total number of subjoins computed for delta compensation is reduced to $2^t - 1$: $\mathbb{J}_{withCache}(t) = \mathbb{J}_{noCache}(t) \setminus \{main\}^t$. For our example from Fig. 4, the subjoin #8 does not need to be recomputed as it is cached. However, all other subjoins in Equation 1 are evaluated during delta compensation.

3. ENTERPRISE APPLICATION CHARACTERISTICS

In this section, we give an overview of enterprise application characteristics, that can be utilized to speedup processing of join queries in the aggregate cache. We have analyzed several enterprise applications including financial and managerial accounting, materials management, and customer relationship management and found out that they all share schema design and workload patterns.

3.1 Schema Design Patterns

In all analyzed application domains, we identified tables with similar design patterns, namely *header*, *item*, *dimension*, *text*, and *configuration* tables.

A *header* table describes common attributes of a single business object. In a financial accounting application, for example, this includes attributes such as the fiscal year and the type of the particular business transaction. In materials management the header tuple stores attributes such as the warehouse origin and destination, and the date and time of a goods movement.

To each header tuple, there are a number of corresponding tuples in an *item* table. Item tuples represent entities that are involved in a business transaction. For instance, all products and the corresponding amount for a sale or materials and their amount for a goods movement are stored in the items table. A header tuple and all corresponding item tuples are also referred to as a *business object* since they are modeled as part of a business transaction.

Additionally, attributes of the header and item tables refer to keys of a number of smaller tables. Based on their use case we categorize them into *dimension*, *text*, and *configuration* tables. *Dimension* tables manage the existence of entities, such as accounts and materials. Especially companies based in multiple countries have *text* tables to store strings for dimension table entities in different languages (e.g., product names). *Configuration* tables enable system adoption to customer specific needs and business processes.

3.2 Application Workload Patterns

According to the table classifications, different workload patterns occur. Not surprisingly, there is a high insert load on tables that contain transactional data (i.e., header and item) compared to dimension, text, and configuration tables.

In many domains, entire *static business objects* are persisted in the context of a single transaction. Therefore, the header and corresponding item tuples are inserted within the same transaction and never changed thereafter. In financial applications, it is even required from a legal perspective that *booked* transaction cannot be deleted, but only changed with the insertion of a counter booking transaction.

In some domains such as customer relationship management and sales, items may be added to a header at a later point in time. This could be the case when a customer adds products to an order. As [24] analyzed a number of enterprise systems, there is only a small amount of updates and deletes compared to inserts and selects on the header and item tables.

Analyzing aggregate queries of the examined applications, a join between header and their corresponding item tuples is very common. Additionally, the analytical queries extract item properties, text strings, and calculation rules from dimension, text, and configuration tables. Those three table categories do have a number of properties in common: There are rarely inserts, updates, or deletes and they contain only a few entries compared to header and item tables.

In the next section, we briefly discuss partition pruning techniques and introduce matching dependencies, and then, in Section 5, we describe how each mentioned enterprise application characteristic can be captured in the application design to allow very efficient query processing with aggregate cache by leveraging the join partition pruning techniques.

4. PARTITION PRUNING AND MATCHING DEPENDENCIES

In this section, we first formally define join pruning for partitioned tables, discuss how these techniques can be applied to columnar tables, and then introduce the concept of matching dependencies which can be leveraged to model and enforce object-aware, temporal relationships.

Each column of a table in SAP HANA is dynamically partitioned into main and delta storages, both columnar stores, hence the columnar tables have a natural mix of vertical partitioning (i.e., columns) and horizontal partitioning (i.e., delta and main). Traditional techniques for partition pruning could be applied to this type of tables during query processing [13, 26]. Formally, horizontal partitioning of a table R is a set of disjoint subsets $\{R_1, ..., R_n\}$ of R such that $R = R_1 \bigcup R_2 \bigcup ... \bigcup R_n$. A table partitioned based on a specified partitioning scheme, must be processed during query execution by accessing each of its partitions based on the query semantics [23]. As some of the partitions may not be relevant to the query, partition pruning methods can be applied to avoid accessing irrelevant data. *Logical partition pruning* refers to methods of pruning based on the definitions of the partitioning scheme (usually applied during query optimization), while *dynamic partition pruning* is a method of pruning based on runtime properties of the data not on the static partitioning scheme (usually applied at query execution time). For dynamic partition pruning, the execution plan can be built with extra physical operators which will allow partition pruning during query execution based on properties which hold for the current instance of the database.

DEFINITION 1. **Join Pair-Wise Partition Pruning** *by a join operator* $\bowtie_q$. *Let* $\{R_1, ..., R_n\}$ *be a horizontal partitioning for a table R. Let* $\{S_1, ..., S_m\}$ *be a horizontal partitioning for a table S. We say that the pair* (R_j, S_k) *is logically pruned by the join operator* $\bowtie_{q(R,S)}$ *if and only if* $R_j \bowtie_{q(R,S)} S_k = \emptyset$ *for any instances of the tables R and S. Let* $\{R^i_1, ..., R^i_n\}$ *be an instance of the table R,* R^i*, and* $\{S^i_1, ..., S^i_m\}$ *be an instance of the table S,* S^i*. We say that the pair of instances* (R^i_j, S^i_k) *is dynamically pruned by the join operator* $\bowtie_{q(R,S)}$ *if and only if* $R^i_j \bowtie_{q(R,S)} S^i_k = \emptyset$.

A simple example of dynamic partition pruning for a join $R \bowtie S$ is pruning all subjoins of the form $R_j \bowtie S_k$ if the partition R_j is empty at the query execution time.

One type of dynamic join partition pruning is based on the range values of the join attributes in each partition (see Example 1). This type of partition pruning is relevant to our solution for addressing performance problems of join queries using the aggregate cache. Note that successful pruning is achieved when the value ranges of the join attributes do not overlap among partitions.

EXAMPLE 1. **Dynamic join partition pruning based on range values.** *Let* $\{R_1, R_2\}$ *be a horizontal partitioning of* $R(A)$*. Let* $\{S_1, S_2\}$ *be a horizontal partitioning of* $S(A)$*. A pair* (R_1, S_2) *is pruned by the join operator* $\bowtie_{R[A]=S[A]}$ *if it can be determined that the instances* S^i *and* R^i *are such that* $R^i_1 \bowtie_{R[A]=S[A]} S^i_2 = \emptyset$.
One runtime criteria for determining that the pair (R^i_1, S^i_2) *is pruned by* $\bowtie_{R[A]=S[A]}$ *could be based on the current range values of the attribute A in the relations R and S. Note that the tuples with NULL value on A will not participate in the join.*
Let $max(R^i_1[A]) = max\{t[A] | t \in R^i_1\}$,
$min(R^i_1[A]) = min\{t[A] | t \in R^i_1\}$,
$max(S^i_2[A]) = max\{t[A] | t \in S^i_2\}$,
$min(S^i_2[A]) = min\{t[A] | t \in S^i_2\}$.
If $max(R^i_1[A]) < min(S^i_2[A])$ *or*
$max(S^i_2[A]) < min(R^i_1[A])$ *then* $R^i_1 \bowtie_{R[A]=S[A]} S^i_2 = \emptyset$.
Proof: If $max(R^i_1[A])$ *and* $min(R^i_1[A])$ *are defined as above, then* $R^i_1 = \sigma_{min(R^i_1[A]) \leq R[A] \leq max(R^i_1[A])}(R)$.
Similarly, $S^i_2 = \sigma_{min(S^i_2[A]) \leq S[A] \leq max(S^i_2[A])}(S)$.
Then $R^i_1 \bowtie_{R[A]=S[A]} S^i_2 =$
$R^i_1 \bowtie_{q(R,S)} S^i_2 = \emptyset$ *with* $q(R, S) = (R[A] = S[A] \wedge$
$min(R^i_1[A]) \leq R[A] \leq max(R^i_1[A]) \wedge$
$min(S^i_2[A]) \leq S[A] \leq max(S^i_2[A]))$
because the join predicate $q(R, S)$ *is a contradiction if* $max(R^i_1[A]) < min(S^i_2[A])$ *or* $max(S^i_2[A]) < min(R^i_1[A])$.

4.1 Matching Dependencies

Matching dependencies are well studied in the literature, for example in [8], and can be used for defining extra relationships between matching tuples of two relations. The matching dependencies extend functional dependencies and were originally introduced with the purpose of specifying matching rules for object identifications [7]. However, matching dependencies can be defined as well in a database system, and can be used to extend functional or inclusion dependencies supported in RDBMSs. They can be used to impose certain constraints on the data, or they can be dynamically determine for a query; they can be used for semantic transformations (i.e, query rewrite), and optimization of the query execution. We adopt here a variant of the definition for matching dependencies introduced in [8].

DEFINITION 2. **A matching dependency** *MD on two relations* (R, S) *is defined as follows: The matching dependency* $MD = (R, S, (q_1(R, S), q_2(R, S))$*, where* q_1 *and* q_2 *are two predicates, is defined as a constraint of the form:*

$$\forall r \in R \wedge \forall s \in S : q_1(r[A], s[A]) \implies q_2(r[B], s[B]) \quad (2)$$

Note that if a matching dependency $MD = (R, S, (q_1(R, S), q_2(R, S))$ holds, it can be used for query optimization, e.g., join pruning, semantic transformations, as the following equality holds for any instance of R and S.

$$R \bowtie_{q_1(R,S)} S = R \bowtie_{q_1(R,S) \wedge q_2(R,S)} S$$

Section 5 details specific matching dependencies defined to model object-aware semantic constraints among tables, and how they can be used for dynamic join pruning for partitioned tables in this context.

5. OBJECT-AWARE JOINS

We discuss in this section some practical design problems of how matching dependencies can be defined, enforced, and used for dynamic join partition pruning as well as join predicate push downs in a RDBMSs. We also discuss how specific semantic constraints among relations can be defined using MDs. While *object-awareness* can refer to various semantic constraints, we focus on temporal locality with regards to record insertion in this paper.

Matching dependencies can be used to impose constraints on two relations which are usually joined together in queries: if two tuples agree on some attributes, then they must agree on some other attributes as well [8]. An example: if two tuples agree on the product attribute, then they must agree on the product category attribute as well. By adding a temporal attribute such as an auto-incremented transaction id, we can use this type of constraint to model temporal locality semantics among relations.

As discussed in Section 3, specific application scenarios have naturally the following semantic constraints among pairs of tables: if a tuple r is inserted in the table R, then a matching tuple s (where $r[A] = s[A]$, $A \subseteq attr(R)$ and $A \subseteq attr(S)$) is inserted in the table S in *the same transaction* as r is inserted, or within *a small range of transactions* from r. To model this type of semantic constraints, MDs can be used. The MDs themselves, as defined here, are strong constraints which are enforced in the database. The constraint that records in related tables are inserted in transactions close to each other, is a temporal soft-constraint. When this temporal constraint holds, using the proposed MDs will guarantee dynamic pruning as matching tuples reside all in delta store or all in main store. If the temporal soft-constraint doesn't hold, the dynamic pruning will not be possible. In both cases, the join pruning using these MDs will be correct. An interesting future work is to model (and dynamically discover) this type of soft-constraints without using strong MDs which require extra storage.

The following design can be imposed to define the MDs between two tables R and S which will allow dynamic partition pruning for join queries using the aggregate cache.

A new column $R[tid_R]$ is added which records the temporal property of the tuples in R as they are inserted into R. We set $r[tid_R]$ to the auto-incremented transaction identifier (generally available in an IMDB) during which the new tuple r is inserted, a value larger than any existing value already in the column $R[tid_R]$. For the table S, which is joined with the table R on the matching predicate $R[A] = S[A]$, a new column $S[tid_R]$ is added which is set, at the insert time, to the value of $R[tid_R]$ of the unique matching tuple in R, if at most one matching tuple exists, e.g. $R[A]$ is the primary key of R. While this does not constrain s to be inserted at a later time than r, the MD captures the temporal relation between matching tuples in r and s. This scenario is used for our benchmarks described in Section 6 for which the corresponding MD defined in Equation 3 holds.

$$\begin{aligned} MD_{R,S} = \; & (R[A, tid_R], S[A, tid_R], (R[A] = S[A]), \\ & (R[tid_R] = S[tid_R])) \end{aligned} \tag{3}$$

In the current prototypical implementation which extends the class of aggregate queries supported by the SAP HANA aggregate cache with join aggregates, MDs are enforced on the application level during record insertion. Theoretically, MDs can be implemented in the database if the database supports general MDs as new type of constraints as proposed in [8]. Then, our specific MDs can be defined as part of the meta data and can be enforced similarly to other constraints such as checking for referential integrity.

5.1 Join Pruning

The matching dependency $MD_{R,S}$ from Equation 3 can be used to perform dynamic pruning for the joins $R \bowtie_{R[A]=S[A]} S$. Let's assume that the tables R and S are partitioned as described in Example 1: $R = (R_1, R_2)$ and $S = (S_1, S_2)$, with S_1 and R_1 containing the most recent tuples of R and S, respectively. Also, the matching dependency from Equation 3 holds. The dynamic pruning described in Example 1 can be attempted. Equation 4 shows the derived join predicate which must evaluate to false for pruning a subjoin.

$$\begin{aligned} & R_1 \bowtie_{R[A]=S[A]} S_2 \\ & \text{using } MD_{R,S} \text{ from Eq. 3} \\ & = R_1 \bowtie_{R_1[A]=S_2[A] \wedge R_1[tid_R]=S_2[tid_R]} S_2 \\ & = R_1 \bowtie_{q(R_1,S_2)} S_2 \\ & \text{where } q(R_1, S_2) \text{ uses min()/max() as in Example 1} \\ & q(R_1, S_2) = \\ & R_1[A] = S_2[A] \wedge R_1[tid_R] = S_2[tid_R] \wedge \\ & min(R_1[tid_R]) \leq R_1[tid_R] \leq max(R_1[tid_R]) \wedge \\ & min(S_2[tid_R]) \leq S_2[tid_R] \leq max(S_2[tid_R]) \end{aligned} \tag{4}$$

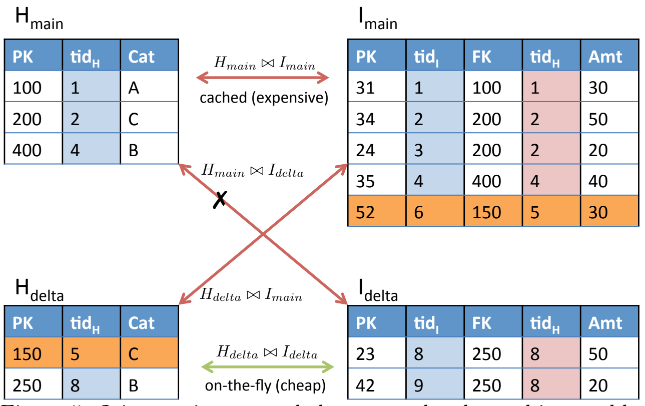
If $q(R_1, S_2)$ can be proven to be a contradiction then $R_1 \bowtie_{R[A]=S[A]} S_2 = \emptyset$. The above technique for dynamic pruning must be done during runtime and it will be always correct as long as $MD_{R,S}$ holds. For example, a prefilter condition defined as in Equation 5, if true, assures that $q(R_1, S_2)$ is a contradiction hence the subjoin $R_1 \bowtie_{R[A]=S[A]} S_2 = \emptyset$ can be dynamically pruned.

$$\begin{aligned} & max(R_1[tid_R]) < min(S_2[tid_R]) \vee \\ & min(R_1[tid_R]) > max(S_2[tid_R]) \end{aligned} \tag{5}$$

In the case of tables in a columnar IMDB, $min()$ and $max()$ can be obtained from current dictionaries of the respective partitions. The pruning will succeed if the prefilter from Equation 5 is true. Otherwise, the pruning will correctly fail if, for example, $MD_{R,S}$ holds but S_2 contains matching tuples from R_1 i.e., the prefilter is false in his case. For an empty partition R_j, we define $min()$ and $max()$ such that the prefilter is true for all join pairs (R_j, S_k).

When the database is aware of the enterprise application characteristics (Section 3) based on their object semantics, join partition pruning can be used to efficiently execute join queries with or without the aggregate cache. We refer to this type of joins as *semantic* or *object-aware* joins.

Let us consider the join query as discussed in Section 2.3 $Q(H, I) = H \bowtie_{H[PK]=I[FK]} I$ joining a header table H and item table I on the join condition, that the primary key $H[PK]$ matches the foreign key $I[FK]$. The matching dependency defined in Equation 6 captures this object-



$H_{main} \bowtie I_{main}$
cached (expensive)
$H_{main} \bowtie I_{delta}$
$H_{delta} \bowtie I_{main}$
$H_{delta} \bowtie I_{delta}$
on-the-fly (cheap)

H_{main}

PK	tid_H	Cat
100	1	A
200	2	C
400	4	B

I_{main}

PK	tid_I	FK	tid_H	Amt
31	1	100	1	30
34	2	200	2	50
24	3	200	2	20
35	4	400	4	40
52	6	150	5	30

H_{delta}

PK	tid_H	Cat
150	5	C
250	8	B

I_{delta}

PK	tid_I	FK	tid_H	Amt
23	8	250	8	50
42	9	250	8	20

Figure 5: Join pruning example between a header and item table with main and delta partitions.

aware semantic constraint, where the attributes $H[tid_H]$ and $I[tid_H]$ are new attributes especially added for the MD:

$$MD_{H,I} = (H, I, (H[PK] = I[FK]), (H[tid_H] = I[tid_H])) \tag{6}$$

The MD is enforced during record insertion in the item table I by setting the attribute $I[tid_H]$ to $H[tid_H]$ of the matching tuple in the header table H. As illustrated in Fig. 5, the item table I has two temporal attributes: $I[tid_H]$ is used to capture the MD with the header table H and $I[tid_I]$ can be used for MDs with other tables that join on the primary key of I.

After an insert into H and I, if there was no merge operation yet, all new matching tuples are in the delta partitions. Therefore, for a delta compensation, we only need to compute the subjoin $H_{delta} \bowtie I_{delta}$ and unify the results with the cached aggregate $(H_{main} \bowtie I_{main})$. Dynamic pruning for the remaining subjoins $H_{main} \bowtie I_{delta}$ and $H_{delta} \bowtie I_{main}$ can be performed if the prefilter condition as defined in Equation 5 holds:

$$max(H_{main}[tid_H]) < min(I_{delta}[tid_H]) \longrightarrow H_{main} \bowtie I_{delta} = \emptyset$$
$$max(I_{main}[tid_H]) < min(H_{delta}[tid_H]) \longrightarrow H_{delta} \bowtie I_{main} = \emptyset$$

Fig. 5 depicts an example of join dynamic pruning for the subjoin $H_{main} \bowtie_{H[PK]=I[FK]} I_{delta} = \emptyset$ as the prefilter $min(I_{delta}[tid_H]) > max(H_{main}[tid_H)$ (i.e., $8 > 4$) is true. However, the subjoin $H_{delta} \bowtie_{H[PK]=I[FK]} I_{main}$ cannot be pruned: the prefilter $max(I_{main}[tid_H]) < min(H_{delta}[tid_H])$ (i.e., $5 < 5$) is false. Fig. 5 highlights the matching tuples in H_{delta} and I_{main} which prevent the join pruning for $H_{delta} \bowtie_{H[PK]=I[FK]} I_{main}$.

5.2 Delta Merge Operation

The incremental maintenance of the aggregate cache takes place during the online merge process which propagates the changes of the delta storage to the main storage [17]. When employing an object-aware join between a header and an item table, if the timing of the delta merge processes could be adjusted for the two tables then the join pruning success rate for delta-compensation and maintenance operations could be maximized. While the dynamic join pruning will always be correct, join pruning is more likely to succeed when the merge processes of related transactional tables are synchronized, rather than when the tables are merged independently, because there is little overlap between delta and main partitions. The example from Fig. 5 shows the case when one of joins $H_{delta} \bowtie_{H[PK]=I[FK]} I_{main}$ cannot be pruned because table I has been merged before H while the join $H_{main} \bowtie_{H[PK]=I[FK]} I_{delta}$ is pruned successfully.

5.3 Join Predicate Pushdown

In case the join pruning does not succeed, we can still leverage the temporal information through the enforced matching dependencies to optimize join processing. Consider the example depicted in Fig. 5, with an overlap of matching tuples in the H_{delta} and I_{main} partitions, which in turn implies that join pruning between H_{delta} and I_{main} cannot succeed.

Based on the matching dependencies, a query optimizer should be able to infer new predicates that can then be pushed down as local filter predicates to the respective partitions, H_{delta} and I_{main}, before evaluating the subjoin. In our example, the subjoin $H_{delta} \bowtie_{H[PK]=I[FK]} I_{main}$ can be rewritten using $MD_{H,I}$ from Eq. 6 and using the runtime domain properties of the attributes $H[PK, tid_H]$ and $I[FK, tid_H]$ follows:
$(\sigma_{f(H)} H_{delta}) \bowtie_{H[PK]=I[FK] \wedge H[tid_H]=I[tid_H])} (\sigma_{f(I)} I_{main})$
with local predicates defined as:
$f(I) = (I[tid_H] >= min(H_{delta}[tid_H]))$ and
$f(H) = (H[tid_H] <= max(I_{main}[tid_H]))$.

Especially the evaluation of $f(I)$ on the I_{main} partition seems to be promising since we do not have to do a full table scan for every potential join partner of H_{delta} but can limit the partition to only consider the relevant records. In the example from Fig. 5, we would only need to check all records for which $f(I) = (tid_H >= 5)$ is true, since $5 = min(H_{delta})[tid_H]$. Similarly, $f(H) = (tid_H <= 5)$, as $5 = max(I_{main}[tid_H]$.

5.4 Applying Join Pruning to Multi Partitions

Up to this point, we have only considered a table to be partitioned into delta and main storage as it is the case in the general main-delta architecture. However, tables can be further partitioned using specific partitioning schemes, for example, as proposed in [25], for *data aging* or *archiving*. We consider a scenario where the columnar tables H and I are partitioned based on the age of the tuples into one *hot* and one *cold* partition. Given the case, that the hot and cold partitioning is static, we can employ a mix of logical and dynamic partition pruning. Thus, the tables H and I each have four partitions $X^c_{main}, X^c_{delta}, X^h_{main}, X^h_{delta}$, where X is any of the tables H or I. There are several interesting properties in this scenario:

- The cold partition X^c_{delta} contains only the updated tuples from X^c_{main} if any. X^c_{delta} is empty in general.
- New tuples are inserted in the hot delta partition X^h_{delta} only.
- The delta-merge operation affects only the hot partition X^h_{main} which is much smaller than $X^h_{main} \cup X^c_{main}$.
- The subjoins on cold and hot partitions of the form $I^c_v \bowtie H^h_w$ with $v, w \in \{main, delta\}$, are always empty, given a consistent aging definition on related tables. These subjoins can be logically pruned. Dynamic pruning can also be applied, almost always, for subjoins between any cold and hot partitions $X^c_v \bowtie Y^h_w$ with $v, w \in \{main, delta\}$.

- There are two aggregate caches defined for subjoins on cold and hot partitions, respectively: $H^c_{main} \bowtie I^c_{main}$, and $H^h_{main} \bowtie I^h_{main}$. The delta-merge operation executed most often affects only the aggregate cache built on hot partitions $H^h_{main} \bowtie I^h_{main}$, hence this is the one which needs to be rebuilt after each merge. The aggregate cache built on cold partitions will be rebuilt very rarely, when tuples are aged into the cold partitions.

To evaluate a query using these aggregate caches, many of the subjoins used for the main-main and delta-main compensation can be pruned. In particular, the subjoins referencing both cold and hot partitions can be partially pruned logically, given a consistent aging definition.

6. EXPERIMENTAL EVALUATION

We first present experimental results for aggregate cache maintenance (in Section 6.1) on the current SAP HANA implementation, performed in a mixed workload of updates and aggregate queries.

Secondly, we assess the performance of query execution without and with aggregate cache for the class of join aggregate queries for which dynamic pruning is performed during delta-compensation. The experimental results are obtained on a prototype implementation which extends the aggregate cache for join queries. For these experiments, we use two benchmarks, the CH-benCHmark [10] based on TPC-H[1] and TPC-C[2], and a benchmark built based on data and workloads from a financial and managerial accounting application of a production ERP system. Opposed to standardized benchmarks such as TPC-C or TPC-H, the second benchmark especially reflects the characteristics of enterprise applications, generating mixed workloads. For this benchmark, the schema contains three tables: a header table *Header* with 35 million tuples, an item table *Item* with 330 million tuples, and a dimension table *ProductCategory* with less than 2000 tuples.

```
SELECT D.Name AS Category, SUM(I.Price) AS
        Profit
FROM    Header AS H,
        Item AS I,
        ProductCategory AS D
WHERE   I.HeaderID = H.HeaderID
        AND I.CategoryID = D.CategoryID
        AND D.Language = 'ENG'
        AND H.FiscalYear = 2013
GROUP BY I.CategoryID
```

Listing 1: Benchmark sample query

We modeled a mixed OLTP/OLAP workload, based on input from interviews and workload traces with an industry customer. The analytical queries simulate multiple users, using a profit and loss statement analysis tool. The SQL statements calculate the profitability for different dimensions including the product category (as mentioned in Section 3) by aggregating debit and credit entries. Listing 1 shows a simplified sample query that calculates how much profit the company made with each of its product categories. The inserts were replayed by using the timestamps in the base data. Deletes and updates were not part of our evaluation workload because they only had a relative low presence in the analyzed ERP production system workloads. All benchmarks were run on a server with 64 Intel Xeon X7560 processor cores and 1 TB of main memory.

[1]http://www.tpc.org/tpch/
[2]http://www.tpc.org/tpcc/

6.1 Maintenance Strategies

We first discuss how our aggregate cache (defined on the main partitions) performs in a mixed workload of inserts and aggregate queries compared to using materialized views with classical maintenance strategies. The statements in this workload reference a single table. Materialized views are defined on main and delta partitions and must be maintained for any delta store changes. Traditional maintenance strategies ensure that a materialized view is always up-to-date when used during query execution: *eager incremental* strategy maintains the materialized views with every insert operation [2], while *lazy incremental* strategy keeps a log of insert operations and maintains the materialized views before it is used [32]. The aggregate cache is defined on the main partition only as presented in this work - and delta-compensation is done at the query time (as shown in Fig. 3). In this experiment, the delta-merge operation is not performed. The insert rates in this experiment bear upon an individual materialized aggregate. In other words, they reflect the number of base data inserts affecting this particular materialized aggregate in relation to the number of times this aggregate is used by read-only queries.

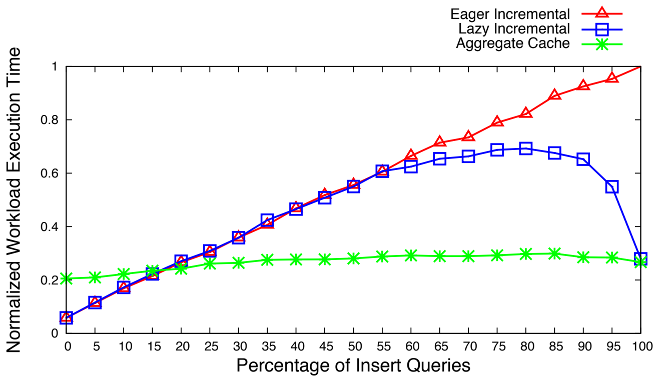


Figure 6: Mixed workload performance using the SAP HANA aggregate cache compared to using materialized views with classical maintenance strategies with varying insert ratios.

The results are depicted in Fig. 6 and reveal that in *write-heavy* scenarios, the materialized view maintenance overhead is very high because materialized views are maintained for any delta changes, either by maintaining the materialized view with every base data modification (eager), or before a read-only query (lazy). Read-only queries using the aggregate cache have an overhead for delta-compensation which is much smaller, in this scenario, compared to the maintenance overhead of materialized views. In a *read-mostly* workload, the materialized view maintenance overhead is marginal as changes do occur very infrequent and the materialized view can directly be used without maintenance by the read-only queries. With an increasing insert ratio however, their maintenance costs increase while our aggregate cache delivers nearly constant execution times due to the fact that the aggregate cache is defined on main stores. Yet, read-only queries using the aggregate cache have an overhead for delta-compensation even if delta store is very small. For insert ratios above 15 percent, this compensation overhead is outweighed by the maintenance overhead by the classical strategies, with the aggregate cache being superior. The shift to a read-mostly overall workload, as described in [25], is not based on number of statements, but on the high percentage of the read-only statements' execution time out of the total workload execution time, which does not necessarily contradict with this experiment.

6.2 Memory Consumption Overhead

In our scenario, we have three tables (header, item, and one dimension table) that need to be extended with the temporal information in order to prune the subjoins. In total, this adds up to the following five additional attributes:

- Header table: $Header[tid_{Header}]$
- Item table: $Item[tid_{Item}, tid_{Header}, tid_{ProductCategory}]$
- Dimension table: $ProductCategory[tid_{ProductCategory}]$

The measured memory consumption, for delta stores, with 2.7 thousand header tuples, 270 thousand item tuples was 78,553 KB compared to 69,507 KB without the temporal information. This is an overhead of 13 percent only for the delta partitions. In main partitions, based on our dataset with 35 million header and 330 million item records, this results in an overhead of 10 percent because of better compression applied to main stores only.

6.3 Insert Overhead

To ensure the matching dependencies of records with foreign keys, every insert operation involving a foreign key attribute needs to find the related *temporal* attribute of the matching tuple. To quantify this overhead, we have measured the time for the look-up of the $Header[tid_{Header}]$ attribute for every insert of a record in the $Item$ table.

The results show that the record insertion in the $Item$ table without the tid_{Header} lookup, and without any referential integrity checks takes about 50 percent of the record insertion time with referential integrity checks. The lookup-up of the matching tid_{Header} value in the $Header$ table takes 20 percent of the time of referential integrity checks. When the number of records in the $Header$ table increase, the look-up slightly increases up to 30 percent. However, this look-up can be combined with the required integrity check for newly inserted records with foreign keys that must find the existing primary key record. Also, we argue that with a shift to a read-mostly workload in enterprise systems [25], the impact of the insert overhead can be regarded as negligible compared to resource-intensive aggregate queries.

6.4 Join Pruning Benefit

To measure the benefit of our proposed join pruning approach, we have created three experiments in which we compare the following four different join query execution strategies:

- *Uncached aggregate query*: this executes an aggregate query without using the aggregate cache as described in Section 2.3.1,
- *Cached aggregate query without pruning*: while the main partition is cached, all remaining subjoins including any delta partitions must be computed for the delta-compensation as described in Section 2.3.2,
- *Cached aggregate query with empty delta pruning*: as an optimization to the previous strategy, we omit subjoins with empty delta partitions as it is the case with the $ProductCategory$ dimension table, and
- *Cached aggregate query with full pruning*: this strategy uses the dynamic pruning concept as described in Section 5.

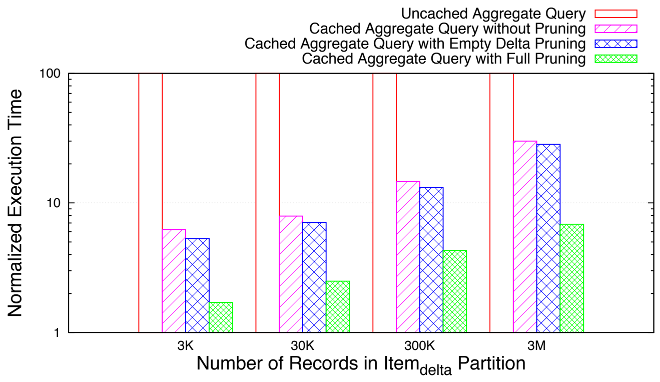


Figure 7: Join performance with different join query execution strategies based on different delta sizes of $Item_{delta}$ and $Header_{delta}$.

The first experiment as illustrated in Fig. 7 measures the execution times of the four different join approaches based on five different delta sizes of the $Item$ table ranging from 300 thousand to 3 million records. The delta partition of the $Header$ table contains approximately one tenth of the $Item_{delta}$ table records and the delta partition of the $ProductCategory$ table is empty. The workload for this benchmark contains 100 aggregate join queries similar to the query in Listing 1. Fig. 7 shows the average normalized execution times of these queries. We see that for small delta sizes, a query using the cached aggregate can be answered by an order of magnitude faster than when not using the aggregate cache. With an increasing number of records in $Item_{delta}$ and $Header_{delta}$ the query execution time increases regardless of the applied join pruning strategy because the newly inserted records in the delta partitions have to be aggregated during the delta-compensation to compute the query results. While the empty delta pruning delivers performance improvements of around 10 percent, the execution times using the full pruning approach is, on average, four times faster than using the cached aggregates without any dynamic join pruning.

In the second experiment, whose results are illustrated in Fig. 8, we have created a mixed workload consisting of insertions of records into $Header$ and $Item$ tables and the execution of aggregate join queries. The starting point is an empty delta partition of both the $Header$ and $Item$ tables. The benchmark then starts the insertion of records in both tables including the look-ups of *tid* attributes. At the same time, we monitor the execution times for aggregate queries executed with the four different strategies. The benchmark has varying frequencies of aggregate queries with respect to the number of inserts which is realistic in an enterprise application context. For example, we can see that there are many aggregate queries at the point of time when $Item_{delta}$ contains around 1 million records.

The results in Fig. 8 show that while the empty delta pruning has minor performance advantages over not pruning

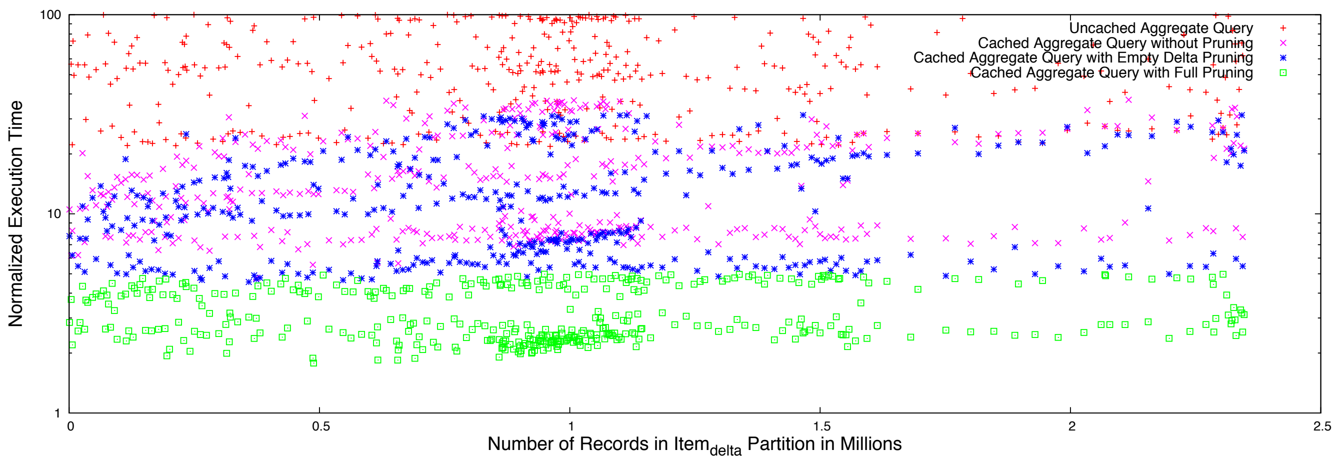


Figure 8: Join performance with different join query execution strategies based on growing delta sizes.

at all, our proposed join pruning approach outperforms both when the delta partitions have non-trivial sizes. We also see that the runtime variance of queries with or without the aggregate cache but without any pruning is very high. This can be explained by a high concurrent system load which, due to the complexity of the monitored aggregate queries, results in variable execution times.

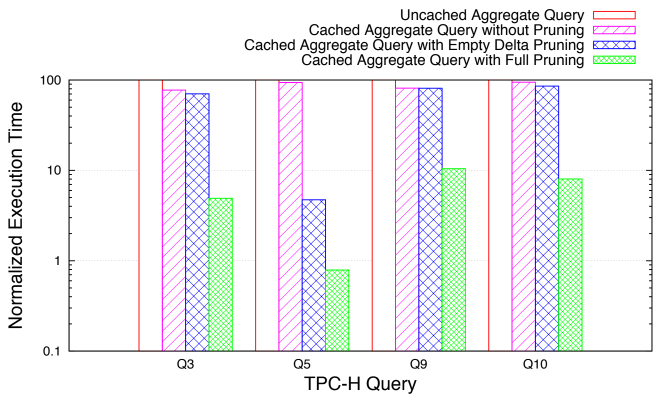


Figure 9: Join performance with different join query execution strategies of TPC-H queries based on CH-benCHmark [10].

As a third experiment for the join pruning benefit, we have taken four analytical TPC-H queries of the CH-benCHmark [10] and analyzed their performance with the four join approaches. The four queries (Q3, Q5, Q9, and Q10) were selected because they are fully supported by the aggregate cache and join more than three tables as in our previous benchmarks. We chose the scale factor 200 for this experiment, which yields 60 million records in the *orderline*, 20 million records in the *orders* table and less records in the remaining tables according to Funke et al. [10]. As proposed in the CH-benCHmark setup, we have populated the delta partitions of the *orders*, *neworder*, *orderline*, and *stock* tables with five percent of total records per table (i.e., the *orderline* table contains 3 million records in the delta and 57 million records in the main), reflecting a mixed workload.

The results are illustrated in Fig. 9 and reveal that for aggregate queries joining more than three tables, the benefit of the aggregate cache is only marginal if delta-compensation during the query execution doesn't use dynamic join pruning. Pruning empty delta partitions yields a minor improvement while the full join pruning approach can accelerate query execution by up to an order of magnitude compared to an uncached aggregate query.

6.5 Join Predicate Pushdown Benefit

In cases when the join pruning is not successful, we can still leverage the temporal relation between the partitions modeled using the MD constraints. In this experiment, we measure the execution time of the subjoins between $Header_{delta}$ and $Item_{main}$ partitions by using the predicate pushdown explained in Section 5.3. We have three different setups with a varying total number of records in $Item_{main}$, while $Header_{delta}$ has a constant number of 100 thousand records. The results as illustrated in Fig. 10 show that with an increasing number of records participating in the join (i.e., matching the join conditions), the performance of the delta-compensation decreases. By using our predicate push-down concept, we can see that it can accelerate the join query execution up to a factor of four, especially if the number of matching records is low compared to the overall table size.

6.6 Applying Join Pruning to Multi Partitions

To benchmark the performance of the join pruning approach in the presence of multiple partitioned tables as outlined in Section 5.4, we have created an experiment with the $Header$ and $Item$ tables, partitioned in a hot-cold ratio of 1:3 as proposed in [25]. We execute five different aggregate queries with different selectivities, aggregating 100 thousand to 25 million records in the hot partition and measure their performance with different join strategies.

The results are illustrated in Fig. 11 and reveal several insights. First of all, we see that an uncached aggregate query is slightly faster in a partitioned environment, because the scan effort can be reduced. Second, we see that the performance of using a cached aggregate query without pruning is worse in a partitioned environment because of the additional subjoins that are required for delta-compensation. The performance of the full join pruning approach is superior in both partitioning scenarios, speeding up query execution by an order of magnitude.

7. RELATED WORK

Materialized views have received significant attention in academia [1, 2, 12, 32], especially in data warehousing envi-

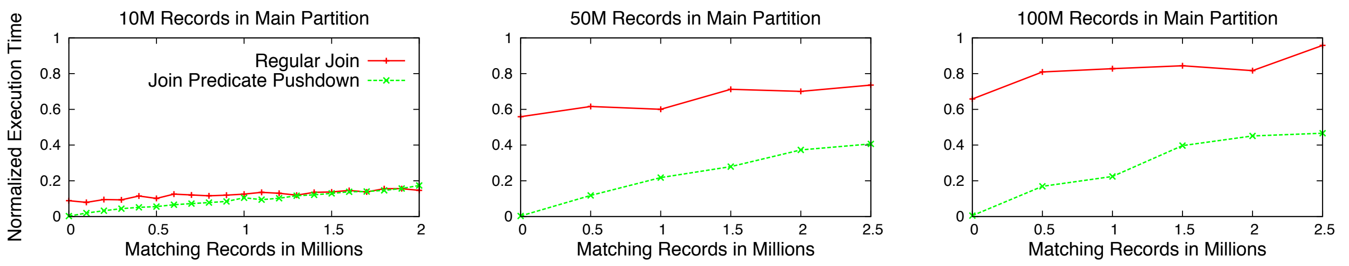


Figure 10: Query execution performance when the subjoin $Header_{delta} \bowtie Item_{main}$ cannot be pruned: with and without the predicate pushdown, based on different $Item_{main}$ partition sizes, and varying in the number of matching records between $Header_{delta}$ and $Item_{main}$.

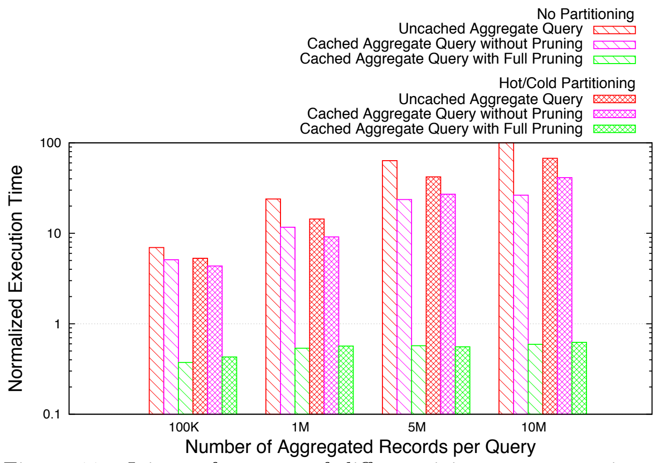


Figure 11: Join performance of different join query execution strategies with unpartitioned and hot/cold partitioned tables. The underlying aggregate queries vary in the number of aggregated records.

ronments [33, 1, 22]. Our techniques for using materialized views are different along multiple dimensions.

First of all, the maintenance timing is not bound to an update of the base data [2], nor it is deferred no later than querying the materialized view [32, 28, 27, 5]. Instead, we maintain the materialized view during the online merge process [17] as our aggregate cache is defined on main partitions. This enables high insert rates and does not imply a maintenance downtime which is not tolerable in mixed workload environments as opposed to data warehouses [4]. Secondly, we do not rely on redundant storage of base data changes, as others do with *auxiliary tables* or *summary tables* [18, 31, 22, 32]. Our delta storage is the primary storage for all inserts and updates performed between two delta merge processes. Our algorithm to calculate the consistent query result of queries using the aggregate cache is similar to the summary-delta tables method introduced in [22], but we do not distinguish between a refresh and propagate phase.

For partitioned tables, several join optimization techniques have been proposed. One of them is to dynamically partition the relations based on workload [26] for improved performance. Another approach is to do logical pruning for horizontally partitioned tables [13]. However, the latter approach is limited to the scenario when the horizontal partitioning attribute matches the join attributes used in the query whereas our implementation supports, by leveraging matching dependency methods, arbitrary join attributes. Also, this approach does not apply to the dynamic partitioning in the general main-delta architecture which we address through dynamic join partition pruning.

While there is an emergence of application-specific databases such as Amazon Dynamo [6] or Google Bigtable [3], we are not aware of a materialized view maintenance and query compensation approach for a general purpose DBMS that leverages the semantics of an enterprise application to increase the performance of aggregate queries using materialized views.

8. CONCLUSIONS AND FUTURE WORK

With the growing requirements of enterprise applications, combining transactional and analytical workloads on a single system, the aggregate cache, a dynamic materialized aggregate engine implemented in SAP HANA, enables the handling of an even higher throughput of aggregate queries generated by multiple parallel users as one mean to scale the system. As admittance in the aggregate cache is directly dependent on the performance of the query execution using the cache, we analyze a special class of aggregate join queries which can be very expensive to compensate. Joins of partitioned tables are challenging in general, but slow down the incremental materialized view maintenance and query compensation of the aggregate cache in particular.

Our analysis of enterprise applications revealed several patterns for their schema design and usage. Most importantly among them, business objects are persisted using a header and item table with additional rather static dimension tables. Moreover, our application workload analysis showed that related header and item records are often inserted within a single transaction or at least within a small time window.

To transport these enterprise application object semantics characteristics into the database, becoming *object-aware*, and optimize the join processing in the aggregate cache, we exploit the concepts of *matching dependencies* and *join pruning* that potentially eliminate expensive joins of partitioned tables. This is achieved by adding temporal attributes at insertion time and use them during run time to dynamically prune subjoins with an empty result set. In addition, we use techniques for *join predicate pushdown*, also based on *matching dependencies*, that can further optimize join processing with aggregate cache when join pruning does not succeed.

The experimental results show that while our approach induces a small overhead for record insertion, the query processing with the aggregate cache using the pruning approach outperforms the non-pruning approach by an average factor of four in the case of three joined tables, and up to an order of magnitude when joining more than three tables or using additional hot and cold partitioning. The join predicate pushdown can optimize a join, in case the pruning does not succeed, up to a factor of four.

One direction of future work includes improving the performance of delta-compensation process for join queries when invalidations are detected in the main storage in case of updates. While the presented join pruning techniques will always deliver correct results, and deletes do not negatively impact the performance of our solution, we are investigating ways to improve the pruning success rate for data updates by keeping track of updates in the delta storage in a separate negative-delta partition. To this end, another interesting future work is to model (and dynamically discover) the temporal soft-constraints among relations without using strong matching dependencies which require extra storage.

9. REFERENCES

[1] D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek. Efficient view maintenance at data warehouses. In *ACM SIGMOD*, pages 417–427, 1997.

[2] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. In *ACM SIGMOD*, pages 61–71, 1986.

[3] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 2008.

[4] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. In *ACM SIGMOD*, pages 65–74, 1997.

[5] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *ACM SIGMOD*, pages 469–480, 1996.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS*, pages 205–220, 2007.

[7] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *Transactions on Knowledge and Data Engineering*, 19(1):1–16, 2007.

[8] W. Fan. Dependencies revisited for improving data quality. In *ACM PODS*, pages 159–170, 2008.

[9] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA database: data management for modern business applications. *SIGMOD Record*, pages 45–51, 2011.

[10] F. Funke, A. Kemper, S. Krompass, H. Kuno, R. Nambiar, T. Neumann, A. Nica, M. Poess, and M. Seibold. Metrics for measuring the performance of the mixed workload CH-benCHmark. In *TPCTC*, pages 10–30, 2012.

[11] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. HYRISE: a main memory hybrid storage engine. *PVLDB*, pages 105–116, 2010.

[12] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

[13] H. Herodotou, N. Borisov, and S. Babu. Query optimization techniques for partitioned tables. In *ACM SIGMOD*, pages 46–60, 2011.

[14] H. V. Jagadish, I. S. Mumick, and A. Silberschatz. View maintenance issues for the chronicle data model. In *ACM PODS*, pages 113–124, 1995.

[15] M. Kaufmann, A. A. Manjili, P. Vagenas, P. M. Fischer, D. Kossmann, F. Färber, and N. May. Timeline index: a unified data structure for processing queries on temporal data in SAP HANA. In *ACM SIGMOD*, pages 1173–1184, 2013.

[16] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *IEEE ICDE*, pages 195–206, 2011.

[17] J. Krueger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. Chhugani, H. Plattner, P. Dubey, and A. Zeier. Fast updates on read-optimized databases using multi-core CPUs. *PVLDB*, pages 61–72, 2011.

[18] W. Lehner, R. Sidle, H. Pirahesh, and R. W. Cochrane. Maintenance of cube automatic summary tables. In *ACM SIGMOD*, pages 512–513, 2000.

[19] S. Müller, L. Butzmann, and H. Plattner. Efficient aggregate cache revalidation in an in-memory column store. In *DBKDA*, pages 66–73, 2014.

[20] S. Müller, R. Diestelkämper, and H. Plattner. Cache management for aggregates in columnar in-memory databases. In *DBKDA*, pages 139–147, 2014.

[21] S. Müller and H. Plattner. Aggregates caching in columnar in-memory databases. In *International Workshop on In-Memory Data Management and Analytics (IMDM), VLDB Workshop*, 2013.

[22] I. S. Mumick, D. Quass, and B. S. Mumick. Maintenance of data cubes and summary tables in a warehouse. In *ACM SIGMOD*, pages 100–111, 1997.

[23] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems.* Springer, 2011.

[24] H. Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In *ACM SIGMOD*, pages 1–2, 2009.

[25] H. Plattner, M. Faust, S. Müller, D. Schwalb, M. Uflacker, and J. Wust. The impact of columnar in-memory databases on enterprise systems. *PVLDB*, pages 1722–1729, 2014.

[26] N. Polyzotis. Selectivity-based partitioning. In *ACM CIKM*, pages 720–727, 2005.

[27] D. Quass and J. Widom. On-line warehouse view maintenance. In *ACM SIGMOD*, pages 393–404, 1997.

[28] K. Salem, K. Beyer, B. Lindsay, and R. Cochrane. How to roll a join: asynchronous incremental view maintenance. In *ACM SIGMOD*, pages 129–140, 2000.

[29] D. Srivastava, S. Dar, H. Jagadish, and A. Levy. Answering queries with aggregation using views. In *VLDB*, pages 318–329, 1996.

[30] J. Wust, M. Grund, K. Hoewelmeyer, D. Schwalb, and H. Plattner. Concurrent execution of mixed enterprise workloads on in-memory databases. In *DASFAA*, pages 126–140, 2014.

[31] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, and M. Urata. Answering complex SQL queries using automatic summary tables. In *ACM SIGMOD*, pages 105–116, 2000.

[32] J. Zhou and P. Larson. Lazy maintenance of materialized views. In *VLDB*, pages 231–242, 2007.

[33] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *ACM SIGMOD*, pages 316–327, 1995.

Transactional Replication in Hybrid Data Store Architectures

Hojjat Jafarpour
NEC Labs America
hojjat@nec-labs.com

Junichi Tatemura
NEC Labs America
tatemura@nec-labs.com

Hakan Hacıgümüş
NEC Labs America
hakan@nec-labs.com

ABSTRACT

We present a transactional and concurrent replication scheme that is designed for hybrid data store architectures. The system design and the requirements are motivated by the real business cases we encountered during the development of our commercial database product. We consider two databases where the original database handles read/write transactional application workloads while the second database handles read-only workloads from the same applications over the data periodically replicated from the original database. The main requirement is ensuring the application of the updates on the replica database in the exact same order they were executed in the original database, which is called execution-defined order. Although this requirement could easily be satisfied by the serial execution of the updates in the commit order, doing so in an efficient manner by exploiting concurrency is a challenging problem. We present a novel concurrency control algorithm to addresses that problem by also allowing the read-only workloads on the replica database to interleave with the concurrent replication. The extensive experiments show the efficacy of the proposed solution.

1. INTRODUCTION

Increasingly more organizations are using multiple database types side-by-side instead of trying to fit one database to all data management needs. The reason is each database product could be better fit for different business requirements. We call the use of multiple database types in the same computing environment hybrid data store architecture.

It is natural that data need to be replicated among those data stores, as the upstream applications ideally would like to use the underlying databases in a seamless fashion without worrying about and the availability of the data sets at a certain location.

An interesting hybrid architecture we are observing is using key-value stores along with relational databases. The relational databases have well known strengths and long, successful history in transactional data processing. Key-value stores have gained popularity for their seamless scalability and elasticity and also their lower-cost profile.

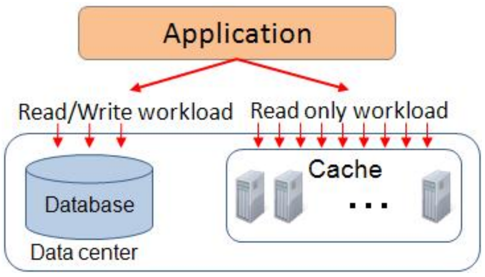


Figure 1: Caching for web applications.

In this work, we specifically focus on the replication, where data need to be replicated from a relational database to a key-value store. The system design and the requirements are motivated by the real business cases we encountered during the development of our commercial database product, Partiqle [20], which is commercialized under the name of IERS[1]. Partiqle is an elastic transactional SQL engine that is implemented on top of a key-value store. In a typical scenario the users already have a traditional relational databases product in use. They want to increase the scalability of the database with the increasing demand from the applications. However, the users don't prefer to scale-up the traditional relational database, instead they consider the scale-out approach through a key-value store, which is the replica of the relational database and serves a specific and demanding part of the workload. Naturally, the main requirement is ensuring the application of the updates on the replica database in the exact same order they were executed in the original database, which is called *execution-defined order*.

Another prominent example of such setting is caching, where data from relational database are cached in a large-scale cache cluster implemented as a key-value store, such as memcached [3]. Memcached is used to significantly reduce the read load on the database system by caching frequently read application generated data in a scale-out in-memory key-value cache. Figure 1 depicts the architecture of a web application that uses memcached. Indeed, most of the largest web applications including Facebook, YouTube, Twitter and Wikipedia are already using memcached as the key scale-out technology in their architecture.

An extreme case of application layer caching approach is

[1]http://www.nec.com/iers

10.5441/002/edbt.2015.58

to cache all the working set data instead of only parts of it. In fact, in this case all the working set data in database is replicated in the cache system that can either be a memory based system like memcached or disk based key-value store system like memcachedb [4] or membase [2] to provide data persistence and recovery. In addition to having the all advantages of normal caching system, such a replication-based system eliminates the cache miss possibility for applications. This is very beneficial when the access pattern to the data is such that reduces the probability of cache hit. Uniform distribution of data access is one of such patterns that may reduce usability of caching. The replication approach also simplifies the application developers' job by not requiring explicit cache value update or invalidation since all the updates are propagated to the replicated data automatically.

With all the benefits of such caching, a major issue that may arise is the exposition of stale data to application, which happens because the replica always lags behind the original data. If there is a high transactional update load on the original data, the replica may significantly be out of sync. Therefore, shortening the lag for the replica would significantly reduce the probability of exposing stale data to the application.

As shown in Figure 1, the relational database handles transactional read/write workload and the key-value store is responsible for handling a read-only workload. The transactional updates in the original database are shipped to the key-value store and applied in the same order to guarantee the correct state for the replica. We call this order as *execution-defined order*. Transactions may interleave during their execution against the original database. However their correct order is defined by their original execution and that order should be respected when doing the log replay for replication. Consequently we have the following requirements:

- The replication process should respect the execution-order of the transactions in the original database.
- The replication process should be efficient to increase replication speed and reduce replica lag.
- The read-only access should be allowed to interleave with an on-going replication process.

The first requirement states that the replication algorithm should guarantee that the resulting serialization order for transactions in the replica is exactly the same as the serialization order in the original database and no other serialization order is acceptable. To illustrate the problem, consider two transactions in Figure 2. If T_i is executed before T_{i+1} then the data item with key Key_k will not exist in the data store. On the other hand, if T_{i+1} is executed before T_i the data item with key Key_k with value $Object$ will exist in the data store. Therefore, although both executions are correct from serialization point of view, the second execution is not acceptable since it does not result in the correct state considering the predefined execution order.

The above requirement could be trivially implemented by replaying the update values in commit order *serially*. However this kind of serial execution would be prohibitively in efficient – the second requirement. Therefore the replication algorithm should improve the efficiency by exploiting concurrency while still respecting the execution-defined order and allowing the read-only workload on the replica to see consistent database state concurrently.

As we discuss in the related work section, although there are related methods in the literature, none of them directly meets the requirements in the given system settings.

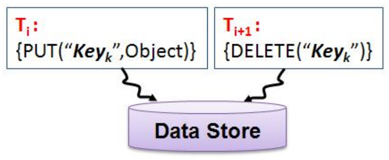


Figure 2: Impact of serialization order on system state.

The contributions of this paper can be summarized as follows:

- We present a replication algorithm that exploits concurrent execution for efficiency while guaranteeing the execution-defined order in the replica database and allowing read-only application workloads to interleave on the replica database.
- We present the architecture of the system, *TxRep*, we implemented based on the replication method presented in the paper and hybrid data store architecture. The actual system is used to generate our experimental results presented in the paper.
- Through extensive experiments we evaluate the performance of the proposed transactional replication algorithm under variety of parameters.

The rest of the paper is organized as follows. We review the related work in Section 2 and then present the system architecture and the consistency model in Section 3. Section 4 describes query translator followed by the details of our transaction manager and concurrency control algorithm in Section 5. We present our experimental results in Section 6. Section 7 concludes the paper.

2. RELATED WORK

Application level caching systems such as memcached [3], memcachedb [4] and membase [2] have been used as scale out solution in many web applications. However, such systems do not provide any transactional consistency guarantees for data access and updates with the rest of the system. Transactional cache (TxCache) provides transactional access to application level caching systems such as memcached [19]. TxCache guarantees that any data accessed regardless of being in cache or database is consistent based on a valid snapshot of the database. It may result in stale snapshots which is acceptable for web applications. Unlike TxCache and other application level caching approaches, our proposed scale out approach replicate whole data base in the key/value store and prevents all read transactions from hitting the relational database.

In the relational database context, a common approach for scale-out is replication [13]. In [17] Manassiev et. al., present a replication technique for scaling and continuous availability of relational databases. The approach assigns a master for each conflict class where all update transactions for the class are sent to the corresponding master. Each master node has a set of slaves that are its replicas and serve the read-only transactions in the system. Updates are disseminated from master to its slave nodes either eagerly upon their arrival or lazily by packing several updates and applying them together. This approach can be further improved by using a modified version of our algorithm in applying the updates on slave nodes.

Providing equivalence to a predefined serialization order has been explored in the context of relational databases. Conservative timestamp ordering guarantees the execution order based on the assigned timestamps to the transactions [10]. By delaying the execution of operations until completion of execution of conflicting operations with smaller timestamp, conservative T/O guarantees there will be no conflicts in execution of each operation. In practice, this approach serializes all write operations in the database. The improved version of this protocol, SDD-1, tries to provide more concurrency by using transaction classes [11]. Transactions are places in transaction classes and only potentially conflicting transaction classes should be dealt with conservatively.

Our concurrency control algorithm is very similar to concurrency control by validation [16]. This approach also assigns three states to transactions, START, VAL and FIN which are equivalent to START, COMITTED and COMPLETTED states in our algorithm. However, in our algorithm we have a predefined order that we should follow while in the validation-based case the order is decided in the validation phase. In the validation-based case if a transaction cannot be validated it is simply rolled back which will result in a different serialization order compared to the case where the transaction could be validated. But in our case this is not acceptable and we strictly should follow the predefined order.

Jiménez-Peris et. al., proposed a deterministic thread scheduling to enable replicas to execute transactions in the same order [14]. This approach requires careful consideration of the impact of interleaving of local threads and scheduled threads.

In [21] Thomson and Abadi propose a distributed database system that guarantees equivalence to a predetermined serial ordering of transactions by combining a deadlock avoidance technique with concurrency control schemes. All the transactions in the system go through a preprosessor component that determines the execution order and then are propagated to replicas using a reliable, totally ordered communication layer. The conflicting transactions are aborted and retried, however, all abort and retry actions are deterministic although the order may change by preprosessor. In our proposed approach, on the other hand, we do not have the possibility of changing serialization order and we should follow the predefined order strictly all the times.

Polyzois and Garcia-Molena proposed a similar algorithm for remote backup in transaction processing systems [18]. To execute transactions in the backup they use tickets to order transactions and two phase locking protocol for concurrency. Each transaction requests lock on the items that it needs, however, the locks are granted according to the transaction ticket number and the protocol ensures that no lock is granted to a transaction unless all the transactions with the smaller ticket that requested the same lock have been granted. Unlike this approach our concurrency control algorithm follows a technique similar to optimistic concurrency control.

3. SYSTEM ARCHITECTURE

Figure 3 illustrates the architecture our implemented system. We assume a standard relational database as the database that stores all the persistent application data. The database system provides a SQL interface for the applications and is responsible for handling read/write transactional workload. It is important to note that, we do not change the standard relational database's API's, query execution mechanisms, or optimizations.

To improve the performance of the database system for certain application workloads, the database is replicated into a distributed key-value store. The key-value store can be memory-based or disk-based store. The replicated key-value store plays similar role to cache for the database and is used to handle the read only workload while the read/write workload bypasses the key-value store and is run directly on the database. The system does not have any specific assumption about the key-value store and as long as the store provides standard PUT/ GET/ DELETE interface to access data, it can be used in our system. For instance, we can use memcached[3], memcachedb[4] or Dynamo [12] as the key-value store in our system. In our system we provide both key-value store API (PUT/ GET/ DELETE operations) and also SQL API to the key-value store. The key-value store API is the native API and it does not require any additional component. To enable SQL API to the key-value store we used our Partiqle system [20]. There are other examples of providing SQL-like APIs to key-value stores, such as UnQL[2] and CQL[3], which could also be used for this purpose.

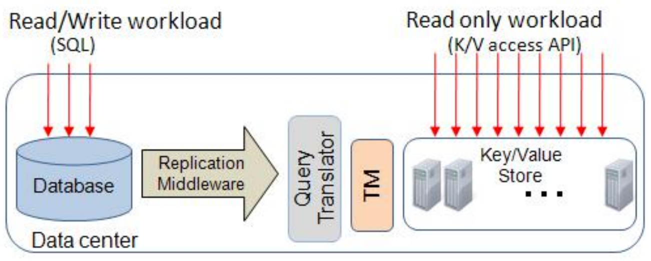


Figure 3: Scale out architecture of TxRep using key-value store.

Between the relational database and the key-value store we have three main components of our system that are responsible for synchronizing the key-value store with the relational database.

The Replication Middleware component is responsible for shipping the transaction log from the relational database to the replica in the key-value store. It periodically reads the transaction log in the database, packs the new transactions into a relocation message and ships the messages to the key-value store. Note that the transaction log only includes write statements and there is no need to apply read statements from the relational database in the replica. We used an MQ-based system (Apache ActiveMQ) as the replication middleware. Although it is an important part of the system

[2]http://www.unqlspec.org

[3]Cassandra Query Language: http://www.datastax.com

architecture, the replication middleware details are not directly relevant for the concurrent replication method, which is the main focus of the paper. Therefore, we give some details of the component in the appendix for the interested readers.

The Query Translator (QT) component is responsible for translating the update only SQL statements into native key-value store API operations that can be directly executed on the key-value store. Note that the replication workload only contains the write operations to key-value store. We will discuss details of the QT component shortly.

The Transaction Manager (TM) component is used to apply the transactions to the key-value store concurrently. The TM component essentially implements the proposed concurrent replication method in the paper. Note that when the transactions reach the TM they are in the form of native key-value store API as they have been translated by QT component. The result of applying the transactions should be exactly the same as applying them in serial manner with *execution-defined order*. We will provide more details on the TM in Section 5.

3.1 Consistency model

Our system provides both transactional and non-transactional access to the stored data. The read/write transactions bypass the key-value store and are run directly on the database. Non-transactional read only workload can be directly executed on the key-value store. This is the same access method that is provided in systems like memcached and memcachedb. For such workload the only consistency guarantee is the one that is provided by the key-value store and the read data may also be stale. Most of the existing key-value store systems can provide key level consistency guarantees where access to single key-value item (a single PUT, GET or DELETE operation) can be atomic.

4. QT: RELATIONAL DATA IN K/V STORE

In this section we present the details of the Query Translator component. We first present the data layout on a key-value store and then describe how transaction logs in SQL format are translated into key-value store API operations. In order to facilitate the discussion, we use a modified version of TPC-W benchmark [8] schema as a running example. Figure 4 depicts the modified relational schema. When a customer orders an item, a new tuple corresponding to the order is added into the order relation.

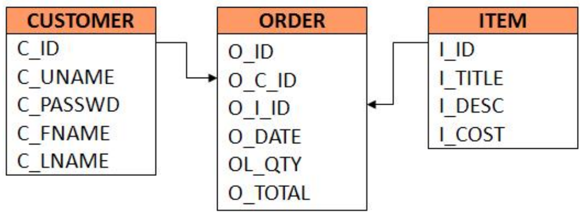
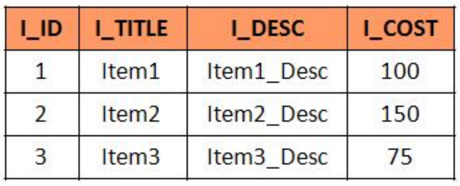


Figure 4: A relational schema for a web based store.

4.1 Relational data over key-value store

Since we replicate the relational data in RDBMS into a key-value store and the data layout on these two stores are different, we need to provide a mapping scheme to map the relational data layout into the key-value data layout.

The first step in storing relational data in the key-value store is to store data in relations. To store the tuples of a relation in the key-value store we represent each tuple as a key-value object. We construct the key for each tuple by combining the name of the relation and the primary key. This generates a unique key for each tuple in each relation. The value for the generated key is the set of all fields for tuple. For instance, consider the ITEM table in our example. Figure 5 depicts three tuples in this table. To represent each tuple as a key-value object, we first create a unique key for each tuple by concatenating the name of the relation with the primary key. For the first tuple the key will be "ITEM_1". The corresponding value for the first tuple will be the set of fields in the tuple, {1, 'Item1', 'Item1_Desc', 100}.



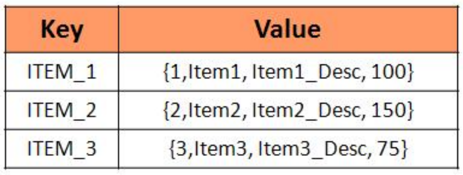
I_ID	I_TITLE	I_DESC	I_COST
1	Item1	Item1_Desc	100
2	Item2	Item2_Desc	150
3	Item3	Item3_Desc	75

Figure 5: Tuples in item table.

The above mapping of relational data into key-value objects provides primary key access to tuples where the application can query, read and write each tuple by its primary key. However, tuples cannot be accessed using any other attribute. For instance, a query cannot access an item based on its cost which is the value of "I_COST" column in ITEM table. This is because there is no key in the key-value store that provides access to item tuples using their cost value. Note that, usually, we are not allowed to scan the entire table: such an operation, which spans over the entire key-value nodes is very inefficient and is not affordable for web applications where the response time is limited. In order to provide access to the tuples through an attribute other than primary key, we create a *hash index* for the attribute in the key-value store. A hash index structure is composed of set of key-value objects. For each distinguished value for the indexed attribute, we create a key-value object. The key is constructed using the value of the indexed attribute and the value for the object is the set of keys for the tuples that the value of their corresponding attribute is the same as the value that was used to construct the key. As an example consider we want to provide access to items through the item cost. For each cost value in the ITEM table we should create a key-value object. The key for such an object is constructed using the relation name, which is "ITEM", the attribute name which is "COST" and the value of the cost column. The value of the object is the keys for the tuples that have the same cost. Figure 7 shows a hash index for the cost attribute in the ITEM relation. Assuming the cost of items with id 1 and 7 is 100, the value for the hash index object with key "ITEM_COST_100" will be "ITEM_1" and "ITEM_7" which are the keys to access the tuples corresponding to these items.

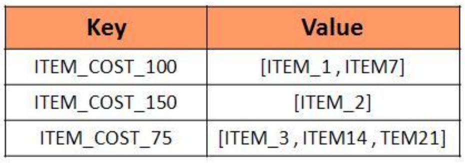
4.2 Range index using B-link tree

Although we can use a hash index to access tuples through any of their attributes, we cannot use it to answer range queries and queries that need to scan a whole table. To pro-



Key	Value
ITEM_1	{1,Item1, Item1_Desc, 100}
ITEM_2	{2,Item2, Item2_Desc, 150}
ITEM_3	{3,Item3, Item3_Desc, 75}

Figure 6: Key/value objects for the tuples in ITEM table.



Key	Value
ITEM_COST_100	[ITEM_1 , ITEM7]
ITEM_COST_150	[ITEM_2]
ITEM_COST_75	[ITEM_3 , ITEM14 , TEM21]

Figure 7: A hash index to access items with their cost value.

vide this capability over key-value store we propose another index structure that is based on B-link tree [15]. B-link tree is a concurrent B-tree that reduces the lock usage for efficiency. A B-link tree is a B^+-tree with an extra pointer in each node. This extra pointer in a node points to the right sibling of the node in the tree. Using this extra pointer, *look up* operation in B-link tree do not need to acquire any locks and *insert* and *delete* operations need to acquire locks on a small number of nodes. We create a key-value object for each B-link tree node. Hence, (1) conflicts among write operations are translated to conflicts on key-value store API operations, which are managed by the TM component (instead of locking), and (2) read-only transactions can access the B-link tree mapped on a key-value store without being blocked by updates.

4.3 SQL statement translation

The replication is done by shipping the transaction log from the relational database and applying the database update operations on the key-value store. Therefore the translation between the relational transactional log to key-value store API operations (such as PUT) involves translating INSERT / UPDATE / DELETE statements from the database log. This operation is not particularly difficult and we used our existing system components from Partiqle system [20] for this purpose.

5. TM: CONCURRENCY CONTROL FOR REPLICATION

The transaction manager (TM) component is responsible for concurrently executing transactions on the key-value store. A transaction starts with *start* statement and ends with *commit* statement. We consider the transactions that are executed by TM are the ones in transaction log of the relational database that were shipped by the replication middleware. However, read-only transactions from application can also be interleaved with the shipped update transactions if they need transactional access to the replicated data in the key-value store. To maintain the correctness of the replicated data in the key-value store the transaction manager should guarantee that the result of concurrent execution of the update transactions shipped from the database is exactly the same as serial execution of them in the same order as they were executed in the database. The simple way to provide such guarantee is to execute all the transactions in the key-value store side serially. However, if the update rate in the database is high, the replica could significantly lag behind the database and increase staleness of the data in the key-value store. It may also significantly reduce the throughput of the read only transactions that are being executed on the key-value store side.

To address this issue we can execute transactions concurrently, however, we need to provide concurrency control to guarantee correctness of the transaction executions. Such concurrency control mechanism is different from the ordinary concurrency control systems because of the execution-defined order of transactions. In an ordinary concurrency control algorithm when a set of transactions are executed, as long as the result of the execution is equal to *some* serial order of transaction execution the result is acceptable. However, in our case the concurrency control algorithm has to guarantee that the result of concurrent execution of transactions is exactly the same as the result of serial execution of them in the same order that they were already executed in the database. Therefore, the existing concurrency control algorithms cannot be used in our TM component.

We propose a new concurrency control algorithm that provides such a guarantee while executes transactions concurrently.

The algorithm receives a set of transactions as input and uses a set of threads in a threadpool to execute the transactions concurrently. Similar to the ordinary concurrency control algorithms we consider two types of conflicts, *read/write* conflict and *write/write* conflict. In read/write conflict two operations conflict if one of them is GET and the other is PUT or DELETE and both access the item with the same key. For instance, the following operations have read/write conflict: $GET(“key1'')$, $PUT(“key1'', object1)$. In write/write conflict two operations conflict if they are PUT or DELETE and both of them access the item with the same key. For instance, the following PUT operations have write/write conflict: $PUT(“key2''\ , object2)\ , PUT(“key2''\ , object3)$.

Two *concurrent* transactions conflict if and only if there is at least one read/write or write/write conflict between their corresponding PUT/ GET/ DELETE operations. Note that if two transactions are not concurrent, i.e., one starts *after* the other completes, they do not conflict even if there are conflicting operations. In order to define concurrency on the key-value store, we assume that the underlying key-value store provides consistent read-write, meaning that a write operation (PUT / DELETE) is atomic and its effect is immediately available for read (GET) operations (no stale data is read). This feature either is supported or can be added easily in most of key-value stores including Dynamo and HBase [12, 1].

Another important assumption that we have is that each transaction is assigned with a sequence number that indicates the place of the transaction in the ordered list of transactions. The sequence number for the update transactions can be easily assigned when the ordered list of transactions are generated by the publisher agent in the database side or when the list is received by the subscriber agent in the key-value store side. However, since we may also want to execute some read only transactions in the key-value store side in a transactional manner we assign the sequence numbers for transactions in the subscriber agent along with assigning sequence numbers to the read only transactions. This guarantees that each transaction has a unique sequence number and the order of sequences for update transactions in the key-value store side is the same as database side while they may be interleaved with read only transactions in the key-value store side.

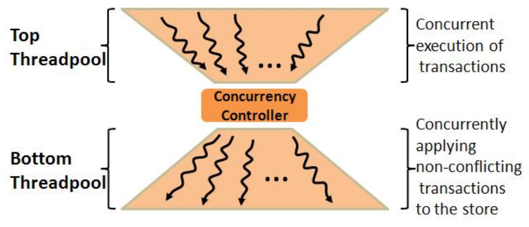

Top Threadpool
Concurrent execution of transactions
Concurrency Controller
Bottom Threadpool
Concurrently applying non-conflicting transactions to the store

Figure 8: Transaction manager component.

Figure 8 depicts the transaction manager (TM) component in our system. There are two threadpools in the system that provide concurrent execution. The first threadpool which is shown on top of the concurrency controller is used for concurrent conversion of transactions into PUT / GET / DELETE operations with their corresponding data items. We refer to this threadpool as *top threadpool*. Each transaction is run over the key-value store using one thread from this threadpool and is represented as a set of PUT / GET / DELETE operations that is going to be evaluated by the concurrency controller. After evaluation of a transaction by the concurrency controller, if there is no conflict, the transaction is passed to the next threadpool that is shown in the bottom of the concurrency controller where another thread is used to apply the results of the transaction to the key-value store. We refer to this threadpool as *bottom threadpool*.

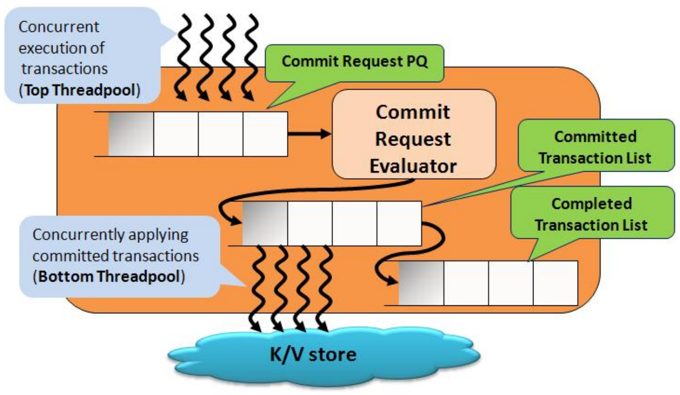

Concurrent execution of transactions (Top Threadpool)
Commit Request PQ
Commit Request Evaluator
Committed Transaction List
Completed Transaction List
Concurrently applying committed transactions (Bottom Threadpool)
K/V store

Figure 9: Internal architecture of concurrency controller.

We now present the details of transaction execution in the transaction manager component. The first step in execution of a transaction in the transaction manager is to detect the set of keys that the PUT / GET / DELETE operations in the transaction access. This is done in one thread that is acquired from the top threadpool shown in Figure 8. For each transaction, we find this set of keys using the query translator and a dedicated buffer. After a transaction starts we create an exclusive buffer for the transaction. For every GET operation in the transaction, if the value for the key does not exist in the buffer the value is retrieved from the key-value store. The value is also stored in the transaction buffer for future accesses. On the other hand, if the value for the key exists in the buffer, the GET operation uses the value in the buffer and does not access the key-value store. For every PUT operation in the transaction the corresponding key-value pair is added to the transaction buffer without accessing the key-value store. Therefore, until the commit statement in the transaction all the changes that are being done by a transaction are stored in the transaction buffer and the transaction does not affect data in the key-value store. In this step multiple transactions execute concurrently using the threads in the top threadpool in Figure 8. When a transaction reaches to its commit statement it is passed to the concurrency controller in the transaction manager. The concurrency controller uses our proposed concurrency control algorithm to detect the conflicts among transactions. If there is no conflict between a transaction and the transactions with lower sequence numbers, the updates of the transaction can be executed concurrently and the key-value store can be updated based on the new values in the transaction buffer. Note that a transaction is only checked for conflicts with its predecessors and there is no need to check for conflict with the transactions that have higher sequence number. If a transaction does not conflict with its predecessors, it can commit by applying its operations to the key-value store using one of the threads in the bottom threadpool as shown in Figure 8. Otherwise, if there is a conflict, because of predefined serialization order the transaction with higher sequence number should restart.

Algorithm 1 depicts our concurrency control algorithm in the transaction manager. Figure 9 also illustrates the internal architecture of the concurrency controller and different data structures that are used by concurrency controller. The algorithm receives the transactions in the form of PUT/ GET/ DELETE operations with the corresponding keys for each operation. These transactions are inserted into a priority queue that is used to sort transactions based on their sequence numbers. After generation of set of PUT/GET/DELETE operations for its corresponding transaction, each thread in the top threadpool puts its transaction in the priority queue. The priority queue, which is referred to as *CommitReqPQ* in the algorithm, is responsible for keeping the order of transactions based on ascending order of their sequence numbers. Each transaction can be in one of the following states:

- ACTIVE: An active transaction has started its execution but has not committed yet.
- COMMITTED: A committed transaction is the one that does not have any conflict with its predecessors. However, the updates in its buffer has not been applied to the key-value store.
- COMPLETED: A completed transaction is a committed transaction that the updates in its buffer have been applied to the key-value store.

In addition, each transaction is assigned with the following values:

- *startTime*: The time that the transaction starts its execution. This is the time when the transaction is assigned to a thread from the top threadpool in Figure 8.

Algorithm 1 Concurrency control in Transaction Manager

```
1: $CommitReqPQ$: Priority queue for commit request.
2: $CommittedTransactionList$: List of committed transactions.
3: $CompletedTransactionList$: List of completed transactions.
4: Transaction $T_i$ is the first transaction in $CommitReqPQ$
5: if $T_i$'s sequence number is NOT the next sequence number. then
6:     goto line 4 (wait for the right transaction).
7: end if
8: Remove transaction $T_i$ from $CommitReqPQ$.
9: for all $Tj \in CommittedTransactionList$ do
10:    if $T_i$ conflicts with $T_j$ then
11:        Add $T_i$ to restart list of $T_j$ ($T_i$ will restarts when $T_j$ is completed.)
12:        return
13:    end if
14: end for
15: for all $T_j \in CompletedTransactionList$ do
16:    if $T_i$.startTime < $T_j$.completeTime then
17:        if $T_i$ conflicts with $T_j$ then
18:            Restart $T_i$.
19:            return
20:        end if
21:    end if
22: end for
23: Add $T_i$ to $CommittedTransactionList$.
24: Change the expected sequence number to $i + 1$.
25: In a new thread from bottom threadpool:
        Execute $T_i$'s statements.
        Move $T_i$ to $CompletedTransactionList$ when the execution is complete.
        Restart the transactions in $T_i$'s restart list.
```

- *commitTime*: The time that the concurrency control algorithm detects that the transaction does not have any conflict with its predecessors.
- *completeTime*: The time that the updates in the transaction buffer for a committed transaction are applied to the key-value store.

The algorithm also uses two lists, one for the committed transactions and one for the completed transactions. The committed transaction list holds the transactions that are in COMMITTED state and have committed successfully. Note that, although these transactions are considered committed, the effect of their execution which is stored in their corresponding buffers have not been applied to the key-value store. The completed transaction list contains the transactions that are in COMPLETED state which are the committed transactions that have also been applied to the key-value store.

The concurrency control algorithm starts by checking the first transaction in the *CommitReqPQ*. If this transaction's sequence number is not the expected sequence number the algorithm does nothing and waits until the transaction with the expected sequence number is put into the CommitReqPQ. If the transaction in the head of queue has the expected sequence number it is removed from the queue and is examined for conflict. Note that when the expected transaction is on top of the CommitReqPQ it means that all the preceding transactions have been evaluated by the algorithm and are in COMMITTED or COMPLETED state. Assuming that the removed transaction is T_i, the algorithm checks the conflicts with the transactions in both *CommittedTransactionList* and *CompletedTransactionList*. The conflict evaluation between T_i and the committed transactions is done in the for loop depicted in lines 9 to 14. If T_i conflicts with a committed transaction T_j, since the changes in T_j have not been applied to the key-value store, T_i may have not seen these changes and therefore, it needs to wait for T_j to apply the changes into the key-value store and restart its execution. In this case, the algorithm adds T_i to the restart list of T_j. The restart list for a transaction is the list of transactions that should be restarted after the transaction is completed and its effect is applied to the key-value store. In case of such conflict since T_i should restart after completion of T_j, the algorithm stops processing other transactions until completion of the conflicting committed transaction. All other transactions after T_i also are not processed since the expected transaction on top of the CommitReqPQ is T_i. After T_j completes, it then notifies all of its conflicting transactions to restart since now they can see the updates from T_j.

If T_i does not have any conflict with the committed transactions, the algorithm checks for conflicts with the completed transactions. This is done in the for loop depicted in lines 15 to 22 in the algorithm. However, as shown in line 16 the algorithm ignores the conflict test between T_i and the transactions that have completed before T_i started. Since we assume writes on the key-value store are immediately available for the readers, there is no concurrency between these transactions: This means that even if there is a conflict between T_i and such transactions, T_i used the updated data from these transactions. On the other hand, if T_i starts before completion of a completed transaction like T_j and T_i conflicts with T_j, then it is possible that T_i may have used out of date data. Therefore, the algorithm restarts T_i in order to make sure that T_i uses the correct data for its execution.

Finally, if T_i does not have any conflict with its predecessors, it can commit and be executed concurrently with them. The algorithm first changes the state of T_i into $COMMITTED$ and adds T_i to the list of committed transactions and updates the expected sequence number. Then, using a thread from the bottom threadpool, it applies the corresponding changes that should be made to data in the key-value store. When the thread finished updating the key-value store, the transaction is completed. At this point, the algorithm changes the state of the transaction to $COMPLETED$ and removes T_i from the CommittedTransactionList and adds it to the CompletedTransactionList. It also restarts all the transactions that have been waiting for completion of T_i. As mentioned, these transactions are stored in T_i's restart list.

5.1 Discarding completed transactions

In our concurrency control algorithm the CompletedTransactionList is the last list that stores transactions. However, by processing more and more transactions this list will grow larger. Therefore, we need to limit the size if this list and remove the completed transactions from the list if there is no need for them. The main reason to keep a completed transaction T_i in the CompletedTransactionList is that if another transaction T_j, starts before the completion of T_i and T_j has conflict with T_i, there is a possibility that T_j did not use the updated data resulted from T_j. Thus, in order to make sure that T_j observes the results of T_i, we need to make sure that T_j starts after completion of T_i. Based on this assumption, if there is no active transaction that has started before completion of a transaction T_i, there is no need to keep T_i and we can safely remove it from the CompletedTransactionList without jeopardizing the correctness of the algorithm.

Algorithm 2 Asynchronous removal of transactions from CompletedTransactionList.

```
 1: $ActiveTransactionList$: List of active transactions.
 2: $CompletedTransactionList$: List of completed transactions.
 3: for all $T_i \in CompletedTransactionList$ do
 4:    boolean shouldBeRemoved = true;
 5:    for all $T_j \in ActiveTransactionList$ do
 6:       if $T_j$.startTime < $T_i$.completeTime then
 7:          shouldBeRemoved = false;
 8:       end if
 9:    end for
10:    if shouldBeRemoved then
11:       CompletedTransactionList = CompletedTransactionList - $\{T_i\}$
12:    end if
13: end for
```

We use an asynchronous algorithm to remove the completed transactions from the CompletedTransactionList. We consider a threshold for the CompletedTransactionList size and whenever the size of the list passes the threshold the transaction removal algorithm is called asynchronously. Algorithm 2 shows the process of detecting and removing completed transactions that are not required from the CompletedTransactionList. For each transaction in the CompletedTransactionList, the algorithm checks if there is any active transaction in the ActiveTransactionList. The ActiveTransactionList is the list that transactions are added when they start execution in the system. If there is at least one transaction that has started before completion of completed transaction T_i, the transaction T_i should not be removed from the CompletedTransactionList. Otherwise, the algorithm removes transaction from the CompletedTransactionList.

6. EXPERIMENTAL EVALUATION

The main goal of our experiments is to validate the advantage of using our proposed concurrency control algorithm and to analyze the effect of different tuning parameters in the performance of the algorithm. In particular we present the following results:

- The comparison of serial execution of transactions with concurrent execution based on our concurrency control.
- The effect of workload characteristics such as conflict ratio, read/write ratio and number of concurrent clients on our proposed concurrency control algorithm.
- The effect of system parameters such as degree of parallelism (number of threads) and key-value cluster size on our concurrency control algorithm.

6.1 Benchmark Description

Since we used web applications as one of the motivating applications for our proposed system, we use TPC-W benchmark [8], which is a transactional web e-commerce benchmark. The benchmark emulates an on-line book store with multiple on line browser sessions. The benchmark provides three different interaction types: browsing (5% of transactions are writes), shopping (20% of transactions are writes) and ordering (50% of transactions are writes). We modified an open source Java implementation of TPC-W benchmark to only emulate database transactions part of the benchmark [9]. The database contains eight tables: customer, address, orders, order line, credit info, item, author, and country. In our implementation we also have two auxiliary tables, shoppingcart and shoppingcartline. These tables are used to store the persistent state of the shoppingcart for each client. We used 2,000,000 items and 2880*1400 (4032000) customers, which results in a database with the size of 7.2GB.

To be able to test the specific parts of the system, we also create a synthetic workload on top of TPC-W database in such a way that we can create transaction conflicts at desired levels. In our synthetic workload each transaction has only one update statement where we update the quantity of an item in the database given the item id. We control the probability of conflict with selecting the item id value from a predefined range. The smaller the selection range, the higher the probability of selecting the same item id for different transactions and therefore, the higher the probability of conflict. Only accessing the same data item is not enough to generate a conflict for two transactions and the other necessary condition is the *concurrent* access to the item by both transactions.

6.2 Experiments Setup

We set up our experimental environment based on our scale out architecture shown in Figure 3. For relational database in the architecture we use MySQL [5] and for the key-value store we use Project Voldemort [6] which is an open source of Amazon Dynamo [12]. We implemented Replication Middleware, Query Translator and Transaction Manager components all in Java. We used Apache ActiveMQ for the messaging middleware in our replication middleware component. The publisher agent reads the transaction log from MySQL and constructs a replication message that is delivered to the replication agent through the ActiveMQ framework. We run the experiments on a set of up to 18 machines. We assign one machine to MySQL database where the publisher agent from the replication middleware also resides on it. Another machine is used as ActiveMQ broker. The Query Translator (QT) and Transaction Manager (TM) along with Subscriber Agent all reside in one machine. The rest of the machines in the experiment are used for key-value store. All the machines except the one that runs the Query Translator (QT) and Transaction Manager (TM) along with Subscriber Agent are Intel Xeon machines with 2.4GHz CPU and 16GB memory running CentOS 5.4. The machine that we used for Query Translator (QT) and Transaction Manager (TM) along with Subscriber Agent has an Intel Core(TM)2 Duo CPU with 3.16GHz speed and 4GB of memory running Ubuntu Linux kernel 2.6.32 and Sun's JDK 1.6.3. In all of the experiments, except the one for evaluating the effect of key-value cluster size, we use five machines for Voldemort key-value cluster.

The metrics that we used in our evaluations are as follows:

Throughput: The throughput is the number of transactions that are executed in one time unit (second).

Execution time: The total execution time is the time it takes to execute all of the given transactions.

Number of Conflicts: As mentioned two transactions conflict when they access the same item concurrently during their execution. For a set of transactions, the number of conflicts is the total number of times that any two transactions conflict during the execution of the transaction set.

We run the workload on MySQL and then use the transaction log to construct the set of transactions with the predefined order that should be applied to the replica in the key-value store. Unless we specify explicitly, in all of the experiments we used 20 threads in top threadpool and 20 threads for bottom threadpool (as shown in Figure 8). The default key-value cluster size for the experiments also is five except for the last experiment.

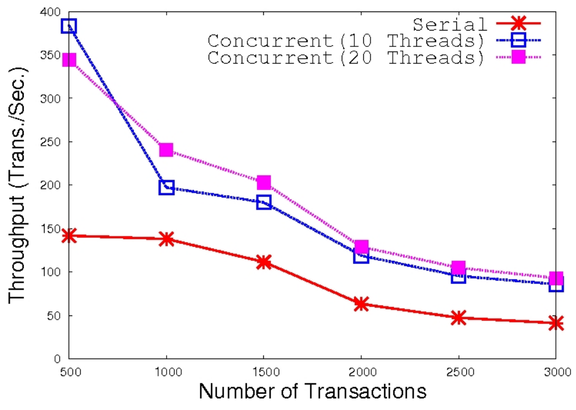


Figure 10: Throughput for Serial and Concurrent execution of transactions.

6.3 Experimental Results

Concurrent vs. Serial execution: The first set of experiments that we present is the comparison of serial execution of transactions with concurrent execution that uses the proposed algorithm. Most of the existing replication approaches use single threaded serial execution of updates in the replica so we use the serial execution as the base line in our experiment [7]. We measured throughput, total execution time and number of conflicts for serial execution and concurrent execution with 10 and 20 threads in our concurrency control algorithm. Figure 10 depicts the throughput for different number of transactions in the replication message that are applied to the key-value store. As it is seen our proposed concurrency control algorithm significantly increases throughput in all cases. Similarly, the total transaction execution time that is plotted in Figure 11 shows that the proposed concurrency control algorithm is at least twice as fast as executing the transactions in serial execution. This will obviously reduce the replica lag behind the original data and consequently the staleness of data in the replica.

As it is seen in both graphs, the benefit of using our concurrency control algorithm is more significant when there are fewer transactions in the replication message. In fact as it is seen, the throughput decrease and execution time increase

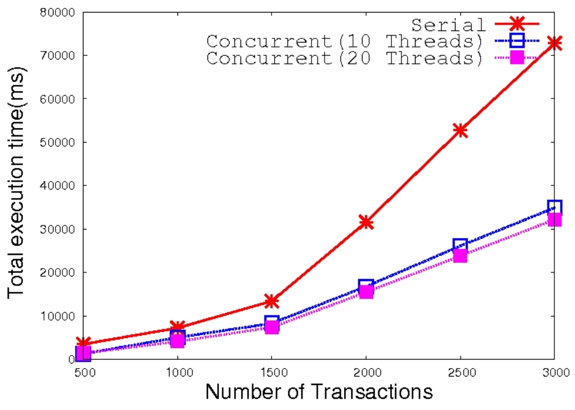


Figure 11: Total execution time for Serial and Concurrent execution of transactions.

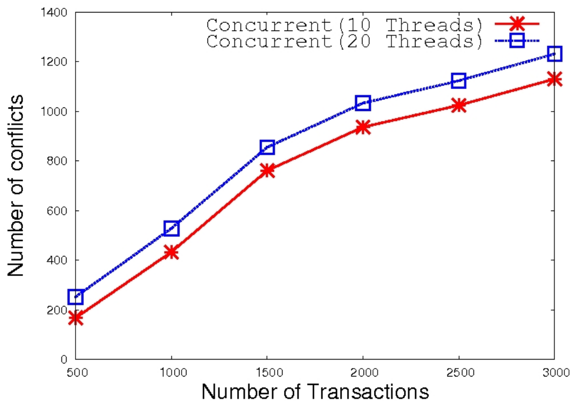


Figure 12: Conflict count for concurrent execution of transactions.

are not linear with respect to the number of transactions that are executed and by increasing the number of transactions in the replication message the throughput reduces faster and execution time increases faster too. This can be described by considering the number of conflicts in the execution process. Figure 12 depicts the number of conflicts occurred in the execution of different number of transactions in the replication message for 10 and 20 threads in the concurrency controller. The number of conflicts increases by increasing the number of transactions. This is expected since the more transactions results in higher probability of accessing same item by multiple transactions concurrently. As described in Section 5, when two transactions T_i and T_j conflict, the concurrency controller aborts the one which is behind in the execution-defined order. Assuming $i < j$, Transaction T_j should be aborted and restarted when the effects of T_i are applied to the key-value store. Note that, this would affect commit time for other transactions by delaying their commit time too. Therefore, as it is seen a conflict will not only slow down the conflicting transactions, but also the ones that are behind them too. Thus, a high number of conflicts reduces throughput more significantly.

Workload Read/Write ratio: We now represent the effect of read/write ratio in the workload on the performance of our concurrency control algorithm. Here read/write ratio is defined as the percentage of write transactions TPC-W interactions. We performed our experiments for all three interaction types in TPC-W and present the results in Table 1. The three interaction types are Browsing where 5% of transactions are write transactions, Shopping where 20% of transactions are write transactions, and Ordering where 50% of transactions are write transactions. The algorithm has better throughput and execution time for browsing and shopping workloads compared to the ordering workload. As we discussed above, the larger number of write transactions increases the probability of conflicts that results in restarting transactions. This indeed increases the number of transactions that are being executed and therefore reduces throughput.

Effect of conflicts: Two transactions conflict if they both access the same data item *concurrently* and at least one of them updates the data item. To evaluate the effect of conflicts in our concurrency control algorithm we use our synthetic workload on TPC-W benchmark where we can control the number of conflicting transactions.

Figure 13 depicts the effect of conflicts on the algorithm throughput. In this figure we plot the improvement percentage over serial execution for 4500 transactions with different number of conflicts. The percentage of improvement in throughput is computed by dividing the difference between the measured throughput for the concurrency control algorithm and the serial execution to the throughput of serial execution and multiplying it by 100. When there is no conflict in the workload we have a steady value for the throughput improvement and the concurrent execution performs twice better than the serial execution (approximately 100% improvement). On the other hand, when we introduce conflicts in the workload the throughput declines as expected. This is caused by restarting the conflicting transactions that slow down the execution of other transactions.

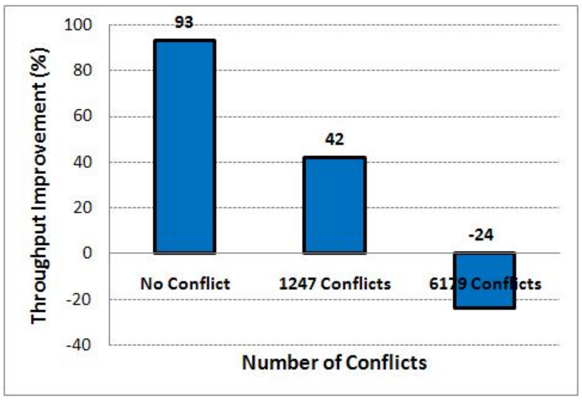


Figure 13: Impact of conflicts in workload on throughput.

The increase of conflicts can have more significant impact on the concurrency control algorithm as it is depicted for the case with 6179 conflicts (transaction restarts) in the graph. In this case, the throughput of concurrent execution is even less than serial execution which does not justify use of concurrent execution for workloads with high conflict ratio. Indeed, we evaluated this in Figure 14 where we show the throughput improvement for different number of conflicts. Similarly, we conclude that the concurrency control algorithm for concurrent execution of transactions is effective only when the number of conflicts in the workload is not too high. In case where the conflict ratio is too high it is better to use the serial execution instead of concurrent execution.

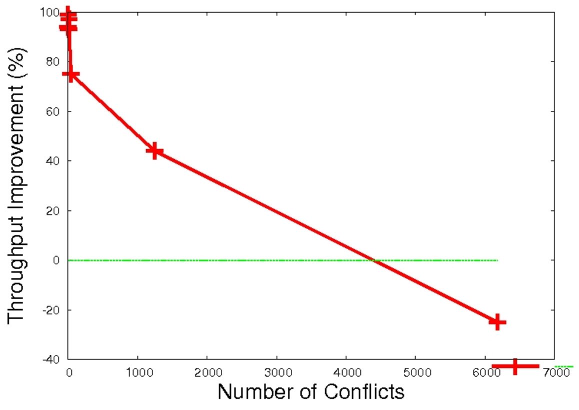


Figure 14: When to use concurrency.

Impact of number of threads in concurrency controller: One of the main tuning parameters in our proposed concurrency control algorithm is the number of threads that are used by the algorithm in initial execution of transactions to construct the set of PUT / GET / DELETE operations and the number of threads to apply non conflicting transactions to the key-value store. The more number of available threads for the transaction conversion to PUT / GET / DELETE operation will result in greater number of transactions requesting to be evaluated by concurrency controller. The larger number of threads for applying non conflicting transactions also should speed up concurrency controller by preventing it from waiting for a non conflicting transaction to be applied to the key-value store. We now present our experimental results on the impact of the number of threads on throughput and the number of conflicts.

Figure 15 plots the throughput of the serial execution along with the concurrency control algorithm for different number of transactions where we use 2, 5, 10 and 15 threads for each threadpool. The overall trend in this graph indicates that by increasing the number of threads we gain higher throughput in our concurrency control algorithm, however, this gain does not increase significantly by adding

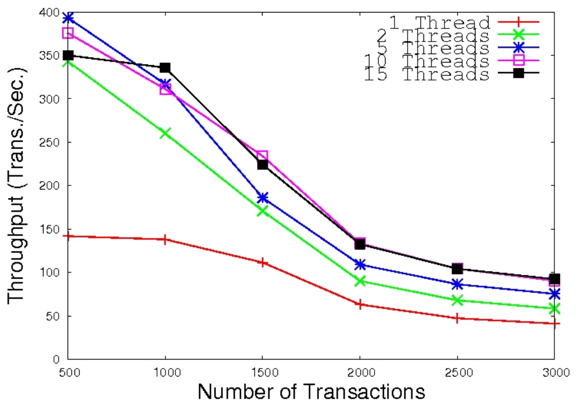


Figure 15: Thread count effect on throughput.

	Browsing (5% write)	Shopping (20% write)	Ordering (50% write)
Write Transactions	200	800	2000
Throughput (Tx/S)	247	256	129
Execution Time (ms)	808	3117	15503
Conflict Count	99	402	1033

Table 1: Results for different TPC-W workloads (4000 Transactions)

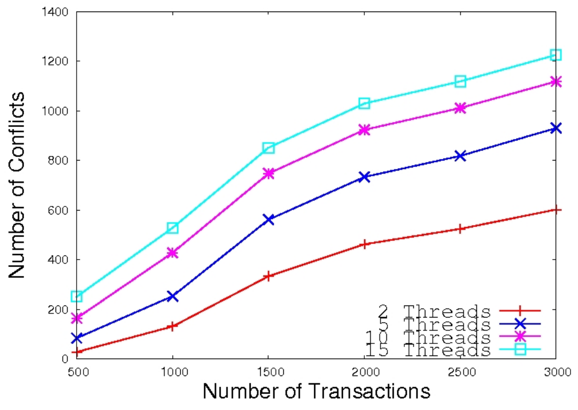


Figure 16: Thread count effect on conflicts.

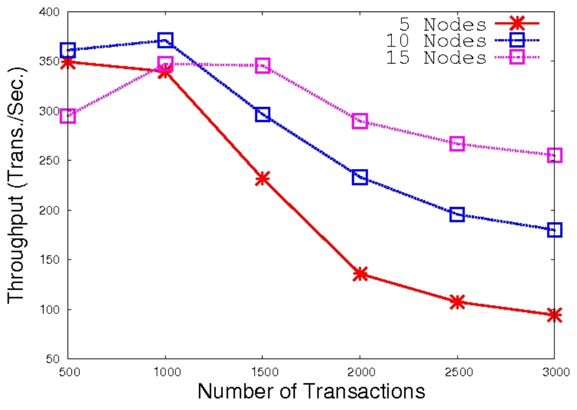


Figure 17: Key-value cluster size.

more threads to the system. As it is depicted the throughput gain for 10 and 15 threads is almost the same. The main reason is that the serial evaluation of transactions for conflicts in concurrency controller. Although we use concurrency in conversion of transactions to PUT / GET / DELETE operations and also in applying the transactions to the store, all the transactions should be evaluated according to their execution-defined order. Therefore, increasing the number of threads can improve throughput initially but at some point the serial evaluation of conflicts in concurrency controller will dictate the execution speed and therefore further increase of the number of threads will have negligible effect on the throughput.

The other factor in reducing the effect of more threads is the increased number of conflicts because of more threads in the system. To have two conflicting transactions not only they should access the same data item where at least one of them writes the data item, but also these accesses should be concurrent. Therefore, increased number of threads elevates the probability of conflict among transactions, which negatively affects the throughput. Figure 16 validates the impact of more threads on the number of conflicts encountered by the algorithm. As shown, by increasing the number of threads in the system, we will have more number of conflicts for the same number of transactions.

Impact of key-value cluster size: In order to analyze the impact of the key-value cluster size on our concurrency control algorithm we used Voldemort key-value store with three different setting, 5, 10 and 15 nodes. Figure 17 depicts the throughput for different number of transactions and different key-value cluster size. The overall trend in the figure is that the throughput is higher when there are more key-value nodes in the system. The larger number of nodes in key-value cluster results in smaller portion of load on each key-value node which in turn speeds up execution of PUT / GET / DELETE operations on each node. Therefore, by increasing the number of nodes in key-value system we can increase the throughput of our concurrency control algorithm.

7. CONCLUSIONS AND FUTURE WORK

We presented a scale out architecture based on fully replication of relational database on key-value store system where the key-value store is used for read-only transactions. Our proposed architecture ships transaction logs from relational database to the key-value store and applies them in such a way that the state of key-value store is exactly the same as the relational database. To reduce the replica lag in the key-value store side we proposed a novel concurrency control algorithm that guarantees a predefined serialization order (the one same as the order in transaction log). We empirically showed that the proposed algorithm significantly improves throughput compared to serial execution of transactions.

An interesting optimization to our concurrency control algorithm is to exploit transaction classes to speed up conflict detection and increase parallelism. By classifying transactions into transaction classes our algorithm would only evaluate conflicts for potentially conflicting transactions. This would eliminate many unnecessary operations and speeds up our concurrency controller.

8. ACKNOWLEDGMENTS

We thank Michael Carey and Hector Garcia-Molina for the insightful discussions and the contributions.

9. REFERENCES

[1] HBase. www.hbase.apache.org.
[2] Membase. www.membase.org.
[3] Memcached. www.memcached.org.
[4] Memcachedb. www.memcachedb.org.
[5] MySQL. www.mysql.com.
[6] Project Voldemort. www.project-voldemort.com.
[7] Replication in MySQL.

http://dev.mysql.com/doc/refman/5.6/en/replication-implementation-details.html.

[8] TPC-W. www.tpc.org/tpcw.

[9] TPC-W Java implementation. http://www.ece.wisc.edu/pharm/tpcw.shtml.

[10] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13:185–221, 1981.

[11] P. A. Bernstein, D. W. Shipman, and J. Rothnie. Concurrency control in a system for distributed databases (SDD-11). *ACM Trans. on Database Systems*, 1980.

[12] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[13] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. *SIGMOD Rec.*, 25:173–182, June 1996.

[14] R. Jiménez-Peris, M. Patiño-Martínez, and S. Arévalo. Deterministic scheduling for transactional multithreaded replicas. In *SRDS*, pages 164–173, 2000.

[15] H. T. Kung and P. L. Lehman. Concurrent manipulation of binary search trees. *ACM Trans. Database Syst.*, 5(3):354–382, 1980.

[16] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6:213–226, June 1981.

[17] K. Manassiev and C. Amza. Scaling and continuous availability in database server clusters through multiversion replication. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '07, pages 666–676, Washington, DC, USA, 2007. IEEE Computer Society.

[18] C. A. Polyzois and H. García-Molina. Evaluation of remote backup algorithms for transaction-processing systems. *ACM Trans. Database Syst.*, 19:423–449, September 1994.

[19] D. R. K. Ports, A. T. Clements, I. Zhang, S. Madden, and B. Liskov. Transactional consistency and automatic management in an application data cache. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–15, Berkeley, CA, USA, 2010. USENIX Association.

[20] J. Tatemura, O. Po, W.-P. Hsiung, and H. Hacıgümüş. Partiqle: an elastic sql engine over key-value stores. In *SIGMOD Conference*, 2012.

[21] A. Thomson and D. J. Abadi. The case for determinism in database systems. In *VLDB*, 2010.

APPENDIX

A. REPLICATION MIDDLEWARE

The data in the key-value store is the replication of the data in the original database. To maintain the replicated data in the key-value store synchronized with the original data in the relational database we use our replication middeware. Details of the replication middleware are depicted in Figure 18. Our replication middleware is implemented on top of a publish/subscribe system. It includes a publisher agent that resides in the database system, a subscriber agent that resides in the key-value store side and a messaging middleware that provides communication framework between publishers and subscribers.The publisher agent periodically reads the transaction log from the database and packages the transactions in a message. The transaction log only includes write statements and does not contain the read statements in the transactions. The frequency of reading the log is a tunable parameter and can be adjusted based on different factors such as staleness limit for read only transactions in the key-value store.

The subscriber agent resides in the key-value store side. This agent receives the messages containing transactions from the publisher agent and applies them to the key-value store through the query translator and transaction manager. Since these transactions have already been executed in the database, the serialization order for the transactions is determined. The easiest way to guarantee such order is to execute these transactions serially over the key-value store. In this case, the subscriber agent issues the transactions to the key-value store through the query translator component using a single thread and each transaction starts after commit of its predecessor. However, as we describe in Section 5 our proposed concurrency control algorithm can guarantee such predefined serialization order while executing transactions concurrently. In this case, the subscriber agent uses a set of threads in a threadpool to concurrently issue the transactions to the key-value store through the query translator and transaction manager components.

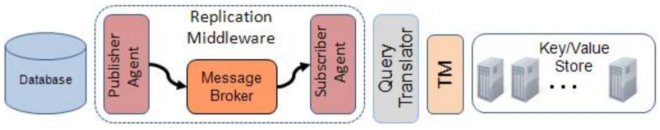


Figure 18: Replication middleware from RDBMS to Key-Value store.

We use a publish/subscribe system to route the transaction log from the publisher agent to the subscriber agent. This functionality is provided by the message broker component as shown in the Figure 18. We consider a single node as message broker, however, a federated set of message brokers can also be used to provide better scalability.

One of the main advantages of using a publish/subscribe system as communication framework for the replication middleware is the decoupling of the publisher agent and the subscriber agent. This eliminates the need for the publisher agent to know the subscriber agent and we can add more subscriber agents to provide multiple replicas without putting any extra load on the publisher agent. All the complexity and load of delivering transaction logs to the corresponding subscriber agents is handled by the message broker and the underlying publish/subscribe system.

SAP HANA –
From Relational OLAP Database to Big Data Infrastructure

Norman May, Wolfgang Lehner, Shahul Hameed P., Nitesh Maheshwari,
Carsten Müller, Sudipto Chowdhuri, Anil Goel
SAP SE
firstname.lastname@sap.com

ABSTRACT

SAP HANA started as one of the best-performing database engines for OLAP workloads strictly pursuing a main-memory centric architecture and exploiting hardware developments like large number of cores and main memories in the TByte range. Within this paper, we outline the steps from a traditional relational database engine to a Big Data infrastructure comprising different methods to handle data of different volume, coming in with different velocity, and showing a fairly large degree of variety. In order to make the presentation of this transformation process more tangible, we discuss two major technical topics–HANA native integration points as well as extension points for collaboration with Hadoop-based data management infrastructures. The overall of goal of this paper is to (a) review current application patterns and resulting technical challenges as well as to (b) paint the big picture for upcoming architectural designs with SAP HANA database as the core of a SAP Big Data infrastructure.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Query Processing

General Terms

Big Data, Streaming, Map-Reduce

1. INTRODUCTION

The event of the "Big Data" hype has triggered a significant push within the data management community. On the one hand, new systems following unconventional system architectural principles have been developed. On the other hand, traditional data management systems have incorporated requirements usually voiced within the context of a "Big Data" discussion. For SAP, the Big Data strategy is of tremendous relevance because of the opportunity to extend traditional as well as reach out to novel application scenarios. A premium example for extending existing applications can be seen with respect to traditional data-warehouse solutions. Many studies show a significant growth in terms of numbers of installations as well as the requirement to embrace non-traditional data sources and data formats [5]. Novel application scenarios address primarily a mixture of traditional business applications and deep analytics of gathered data sets to drive business processes not only from a strategic perspective but also to optimize the operational behavior.

With the SAP HANA data platform, SAP has delivered a well-orchestrated, highly tuned, and low-TCO software package for pushing the envelope in Big Data environments. As shown in figure 1, the SAP HANA data platform is based in its very core on the SAP HANA in-memory database system accompanied with many functional and non-functional components to take the next step towards mature and enterprise ready data management infrastructures. In order to be aligned with the general "definition" of Big Data, we outline the SAP HANA data platform capabilities according to the criteria volume, velocity, and variety.

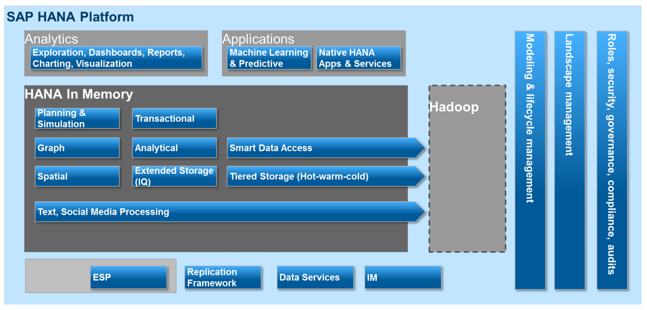

SAP HANA Platform
Analytics
Exploration, Dashboards, Reports, Charting, Visualization
Applications
Machine Learning & Predictive
Native HANA Apps & Services
HANA In Memory
Planning & Simulation
Transactional
Graph
Analytical
Smart Data Access
Spatial
Extended Storage (IQ)
Tiered Storage (Hot-warm-cold)
Text, Social Media Processing
Hadoop
ESP
Replication Framework
Data Services
IM
Modeling & lifecycle management
Landscape management
Roles, security, governance, compliance, audits

Figure 1: SAP HANA Data Platform Overview

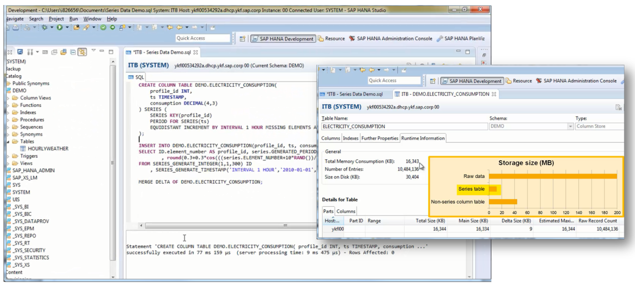
Variety

Since the SAP HANA database system has its origins partially in document processing, support for semi- and unstructured text is part of SAP HANA's DNA [8]. Recent extensions to cope with the aspect of variety additionally consists in comprehensive support for geo-spatial data and time series data as seamless extensions of the traditional table concept. Moreover the pure relational concept is relaxed and extended within SAP HANA within two directions. First, SAP HANA offers so-called "flexible tables" to extend the schema during insert operations allowing applications to extend the schema on the fly without the need to explicitly trigger DDL operations. Second, SAP HANA provides a native graph engine next to the traditional relational table engine to support schema-rich and agile data settings within one single sphere of control and based on the same internal storage structures [22]. Such an architecture reduces TCO by operating only one single system and–at the same time–allows for cross-querying between different data models within a single query statement.

Figure 2 shows an example of time series support for a rich set of

10.5441/002/edbt.2015.59

series data style application scenarios e.g. within monitoring manufacturing equipment or analyzing energy meter data in the large. As can be seen, the system does not only provide the opportunity to explicitly model the semantics of the data set (e.g. certain characteristics or missing value compensation strategies) but also provides an optimized internal representation to increase query performance and reduce the memory footprint. As indicated in figure 2 it is quite common to compress the data by more than a factor of 10 compared to row-oriented storage and more than a factor of 3 compared to columnar storage.



```
CREATE COLUMN TABLE DEMO.ELECTRICITY_CONSUMPTION(
    profile_id INT,
    ts TIMESTAMP,
    consumption DECIMAL(4,3)
) SERIES (
    SERIES KEY(profile_id)
    PERIOD FOR SERIES(ts)
    EQUIDISTANT INCREMENT BY INTERVAL 1 HOUR MISSING ELEMENTS A
);

INSERT INTO DEMO.ELECTRICITY_CONSUMPTION(profile_id, ts, consum
SELECT ID.element_number AS profile_id, series.GENERATED_PERIOD
        , round(0.3+0.3*cos(((series.ELEMENT_NUMBER+10*RAND())/
FROM SERIES_GENERATE_INTEGER(1,1,300) ID
    , SERIES_GENERATE_TIMESTAMP('INTERVAL 1 HOUR','2010-01-01',

MERGE DELTA OF DEMO.ELECTRICITY_CONSUMPTION;
```

```
Statement 'CREATE COLUMN TABLE DEMO.ELECTRICITY_CONSUMPTION( profile_id INT, ts TIMESTAMP, consumption ...'
successfully executed in 77 ms 159 µs  (server processing time: 9 ms 475 µs) - Rows Affected: 0
```

Table Name: ELECTRICITY_CONSUMPTION
Schema: DEMO

General
Total Memory Consumption (KB): 16,343
Number of Entries: 10,484,136
Size on Disk (KB): 30,404

Details for Table

Host:...	Part ID	Range	Total Size (KB)	Main Size (KB)
ykfl00			16,344	16,334

Figure 2: SAP HANA timeseries support

Velocity

In order to cope with the requirements coming from the "velocity" dimension of the Big Data definition, the SAP HANA data platform addresses velocity from an end-to-end perspective. With SQL Anywhere, the platform provides a mature instrument for distributed data capturing, local processing, and efficient propagation into central database installations. As can be seen in figure 1, external data can be imported using Data Services instruments [2] and replicated using SAP Replication Server [3]. In order to cope with extreme low-latency requirements, the mature event stream processor (ESP) solution is integrated and coupled with the in-memory database system. Section 3 will provide a deeper look into this technical solution.

Volume

While volume is fundamental to "Big Data", the SAP platform implements the full spectrum ranging from being able to work with large main-memory installations as well as providing a disk-based extension based on Sybase IQ technology. While the core in-memory HANA engine has successfully proven to work on systems with 12 TByte in a single appliance configuration (e.g. HANA-Hawk [18]), Sybase IQ still holds the world record for the largest DW installation with 12.1 PByte [21]. The transparent integration of IQ technology into the HANA core database system yields an unprecedented combination of high performance as well as the ability to handle large volumes, well beyond of any of today's enterprise-scale requirement.

Application Patterns

Looking at the different requirements, the SAP HANA infrastructure is designed to cope with different application patterns ranging from traditional decision support systems to web-scale operational recommendation applications as shown in figure 3. Typically, data from online sources is captured and pre-filtered using the SAP HANA ESP component and then forwarded to the central SAP HANA system. In addition, low-level log data stored in Hadoop infrastructures are tapped and used for statistical algorithms (e.g. recommendation algorithms). Staying within the single system, the outcome is directly forwarded to the operational system (e.g. SAP ERP) to trigger a subsequent business request.

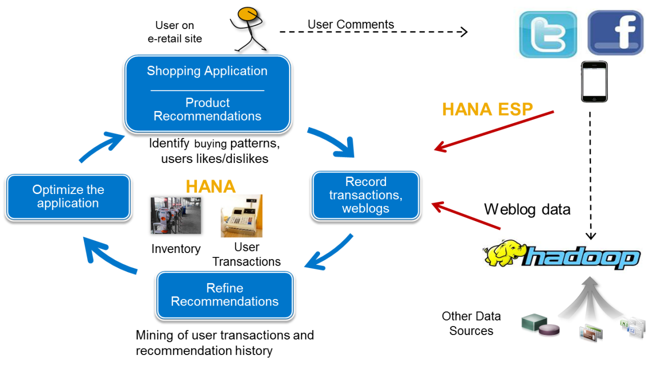


Figure 3: SAP HANA Big Data Infrastructure Core Components

In order to cope with such very typical application patterns, a comprehensive solution acting as an umbrella for individual systems and algorithmic tasks is required, which may consist of different components but presenting itself as a single point of control for the application as well as administration.

2. BEYOND THE TRADITIONAL Vs

While the SAP HANA data platform is well setup to cope with the traditional V-requirements, SAP especially focuses on delivering additional **V**alue to customers and therefore goes beyond the traditional Vs.

Value

The SAP HANA platform provides added **V**alue scenarios for customers by representing not only a core database system but an enterprise-scale data platform with additional services like:

- integrated repository of application artifacts for holistic life cycle management; for example application code in combination with database schema and pre-loaded content can be atomically deployed or transported from development via test to a production system.
- single administration interface and consistent coordination of administrative tasks of all participating platform components; for example, backup and recovery between the main-memory based SAP HANA core database and the extended IQ store is synchronized providing a consistent recovery mechanism.
- single control of access rights based on credentials within the platform; for example, a query in the SAP HANA event stream processor (ESP) may run with the same credentials as a corresponding query in the SAP HANA core database system.

More technically, SAP HANA defines **V**alue of data with respect to relevance distinguishing the low and high density data being handled with different technologies embedded into the SAP HANA platform. Figure 4 outlines the interplay between age (on the x-axis) and **V**alue (on the y-axis). Very recent data may come into the system at a high speed and high volume and is directly digested by the HANA Streaming component (ESP); from there, enriched and

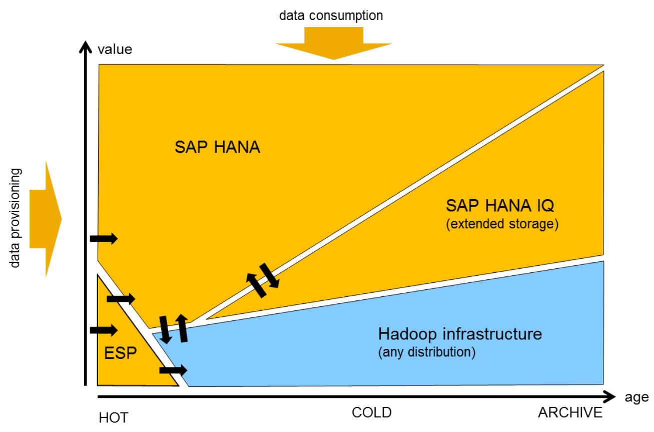


Figure 4: SAP HANA Big Data Infrastructure Core Components

potentially aggregated data can be propagated into the core SAP HANA database engine. In low-velocity scenarios raw data may directly be loaded into the SAP HANA database or handed over to a Hadoop installation as cheap background store for low-density data. High **V**alue is defined as enriched data after usually complicated transformation and cleansing steps. For example "golden records" within a master data management system or statistically corrected and improved fact data within a typical data warehouse scenario represent data sets with high relevance and customer data. As this intensively used data usually fits into the main-memory of a smaller cluster of machines, this data can be managed by the SAP HANA core database engine. HANA low-density data or data with low business value reflect fine-grained log or sensor data, (frozen) archives for legal reasons or extracted images of source systems required to feed complex data transformation processes. For economical reasons but also because of its high volume archival data is stored on disk. **V**alue therefore does not directly correlate with volume, i.e. even cleansed business data may show–especially considering historic data–significant volume but require extremely high performance. Hence, depending on its value we store high-volume data with high-value data in the extended storage and low-value data in Hadoop.

SAP Big Data = Data Platform + Applications

As mentioned before, the SAP HANA platform provides a single point of entry for the application as well as reflects a single point of control with respect to central administration for backup/recovery or life cycle management. Therefore, SAP defines Big Data infrastructures not only via core technology components but as a comprehensive application suite with support for data capturing, cleansing, transformation, and finally consumption using a large variety of transactional and analytical applications. From that perspective, the SAP business warehouse (SAP BW) solution may be considered a foundation and starting point for Big Data Analytics using the SAP HANA platform. The SAP BW solution comprises a complete tool chain for data provisioning and consumption in the traditional data warehouse sense and is based (in addition to other database systems) on the SAP HANA core database engine. Sybase IQ may be used (alternatively to other approaches) as nearline archive controlled by the SAP BW suite. Also, as shown in figure 5, SAP BW (starting with version SPS08) may exploit the extended storage to transparently move cold data to the SAP HANA extended storage. The SAP Big Data Strategy is now pushing the envelope in an evolutionary way with respect to streaming and extended storage as well as supporting Hadoop installations and without compromising the industry-strength tool chain of data provisioning as well as data consumption.

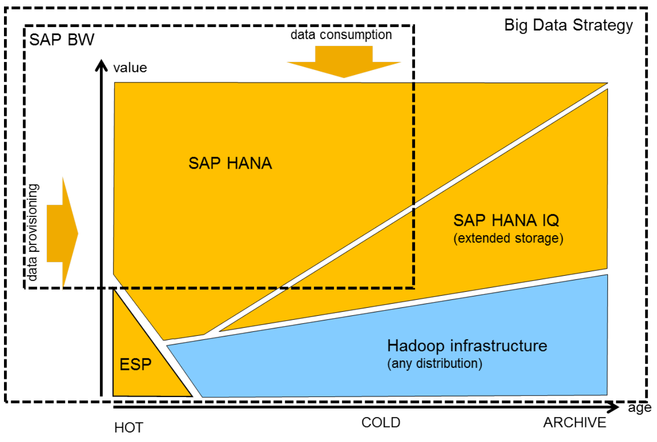


Figure 5: Extension of SAP HANA Business Warehouse

SAP Big Data–The Native and Open Strategy

Figure 5 sketches the positioning of the SAP data platform from a specific perspective of the overall Big Data design space. Obviously, the data platform is designed to deploy the components required to solve a specific Big Data problem using SAP proprietary as well as embracing open systems. In general, this leads to the following SAP Big Data strategy:

- **HANA Native Strategy:** For any Big Data installation, a data platform with only native SAP HANA components is able to cope with any number of volume, velocity, and variety requirements ([8]).
- **HANA Open Strategy:** Any existing Hadoop installation[1] can be embedded into a SAP HANA ecosystem leveraging SAP HANA added value capabilities like life cycle management and integrated application development.

Following these statements, we will detail the technical implications in the remainder of this paper. Section 3 sketches the integration of SAP Sybase IQ as extended storage (HANA IQ) natively into the SAP HANA core database engine and outlines the native support of SAP Sybase ESP as streaming component with SAP HANA (HANA Streaming). Following this native integration concepts, we will outline challenges and opportunities of the HANA open strategy and show the integration of data residing in the Hadoop Distributed File System (HDFS) and pushing processing logic to Hadoop (code to data).

3. HANA NATIVE INTEGRATION POINTS

As outlined, the native Big Data strategy of SAP addresses the full spectrum of Big Data use cases using the native SAP HANA in-memory storage together with a native integration of the IQ storage and ESP event streaming systems. The SAP HANA core in-memory engine is optimized for processing large volumes of data in an OLTP and OLAP-style allowing to serve as a consolidation vehicle for transactional as well as decision support systems [19].

[1]Currently, distributions of Hortonworks are preferred.

Within a Big Data installation, the SAP HANA core database system is usually positioned to keep high value data, i.e. hot and warm data for high-performance access. Although main memory prices went down dramatically in the recent past, it is still not cost-effective to keep bulks of cold data or archival data in main memory [9]. The SAP HANA data platform addresses this economical aspect through native integration points with the extended storage concept which is based on the IQ storage engine. However, in order to provide an end-to-end solution for Big Data scenarios, the SAP HANA data platform integrates tightly with the HANA event stream processor (ESP) of SAP for high-velocity data of (yet) low value.

Other database vendors also attempt to optimize the storage technology for the access patterns of the data. For example, the DB2 multi-temperature feature [1] allows to assign hard disks or SSDs as storage technology on the level of table spaces. Also, DB2 BLU can be used to opt either for conventional row-based and disk-based processing or column-oriented (and mainly) in-memory processing on the table level [20]. In a similar fashion, Microsoft SQL Server also features the in-memory engines Hekaton [7] and Apollo [13] for in-memory processing of data. In [23] the authors discuss a prototype for the in-memory row store, Hekaton, for identifying tuples that can be stored in cold storage. Overall, for both DB2 and SQL Server the decision for in-memory processing currently seems to be done on the table level while SAP HANA supports hot and cold partitions within the same logical table.

As we discuss in this section, both extended storage and ESP are integrated into the SAP HANA query processing. On the application level, a common repository is used to manage the life cycle of development artifacts. Administrators and developers use the SAP HANA Studio as the main interface for development and administration tasks. Meta data is centrally managed by SAP HANA and exploited during query optimization and query execution. Finally, aspects like backup and recovery are tightly integrated. In the following, we will outline technical details of this integration.

3.1 HANA IQ extended storage

SAP HANA offers a variety of options to store data: row-oriented storage in main memory is used for extremely high update frequencies on smaller data sets and the execution of point queries. Column-oriented storage is the basic option for both read- and update-intensive workload, especially for scan-intensive OLAP workloads. In this section we discuss a new storage option, the *extended storage*, which addresses use cases where massive data sets are mainly inserted and infrequently used for reporting. According to [9], it is still the case that rarely accessed data shall primarily reside on (cheaper) disk storage. In order to address this requirement, the extended storage option in SAP HANA offers a tightly integrated disk-based storage based on the Sybase IQ storage manager [15].

For example a concrete scenario for using the extended storage related to the SAP BW application includes the management of the persistent staging area (PSA) where massive amounts of data from a source system are directly mirrored into the BW infrastructure. The main purpose of a PSA is to keep the extracted data sets as sources for refinement processes within BW as well as for provenance and archiving reasons. Since objects of a PSA are (after being used to derive higher value data sets by different BW processes) only rarely used, a disk-based storage reflects an economically reasonable solution without compromising the overall in-memory approach of SAP HANA.

A similar use case with respect to the management of a PSA are so-called write-optimized DataStore objects (DSO) which serve as a *corporate memory*, i.e. data must be kept for very long durations for legal reasons. Similar to PSA objects, DSOs are rarely accessed and performance is not of paramount concern. Overall, using the extended storage SAP HANA customers transparently enjoy the benefit of low-cost storage on disk, but they can still expect reasonably short response times for infrequently accessing data stored on disk.

As a natural extension for hot and cold data stored in a single table, *hybrid tables* allow one or more partitions of a table to be represented like a SAP HANA in-memory column table and other partitions as extended storage. Figure 6 outlines the architectural setup. The IQ engine is completely shielded by the SAP HANA environment and therefore not accessible to other applications. This restriction allows for a tight integration with SAP HANA as the only "application" for IQ. In addition to query shipping to external partitions, SAP HANA provides an integrated installation and administration using SAP HANA Studio. Additionally, the tight integration allows to provide a consistent backup and recovery of both engines. Furthermore, the system allows to directly load mass data into the extended storage and register the data at the orchestrating SAP HANA instance. This direct load mechanism allows therefore to support Big Data scenarios with high ingestion rate requirements.

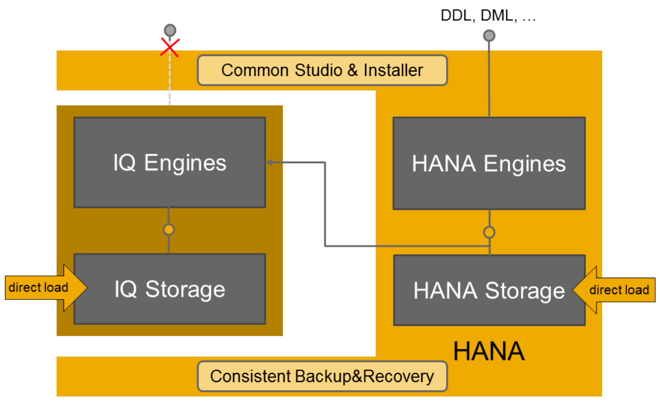


Figure 6: SAP HANA IQ Extension Overview

In summary, the concept of hybrid tables spanning SAP HANA and the Sybase IQ storage manager has the following benefits:

1. It provides a simplified system landscape as SAP HANA is the only interface to both hot and cold data including simplified setup or upgrade as well as integrated backup and recovery.

2. Seamless growth of the data can be managed in the data warehouse far beyond available main memory.

3. Integrated query processing with the extended storage including function shipping to the extended storage exposes the native query performance of the IQ storage engine even when using as extended storage orchestrated by SAP HANA.

In addition to extending SAP HANA for dealing with cold data, the extended storage deployment scenario also provides a seamless migration path from standalone IQ installations to the SAP HANA data platform. Without moving data out of an IQ system, an IQ instance can be registered at a SAP HANA system, and the customer may take advantage of the overall SAP data platform properties.

Extension on Table and Partition level

In using the extended storage, two different levels of working with the extended storage option can be differentiated: In the first scenario, an existing Sybase IQ table is directly exposed and therefore accessible in SAP HANA using the following syntax:

```
CREATE TABLE table_name table_definition
USING HYBRID EXTENDED STORAGE
```

In this syntax the HYBRID clause is optional, because the extension is based on a table level and no mixture of IQ and SAP HANA tables is configured [4]. The extended storage technology performs data type mappings between the engines as well as provides full transactional support. Additionally, a data load issued against such an external table directly moves the data into the external store without taking a detour via the in-memory store followed by a subsequent push into the extended store. Moreover, the extended storage technique supports schema modifications like any other table in SAP HANA to complete the hybrid table concept.

In the second scenario of partition level extension, an SAP HANA table may be partitioned into one or multiple hot partitions, which are realized by regular in-memory column-oriented storage and one or multiple cold partitions which live in a table of the IQ storage. Such a resulting table is considered a SAP HANA hybrid table. The residence of hot and cold data is decided on the partitioning criteria and can be controlled by the application. Additionally, the SAP HANA data platform provides a built-in aging mechanism, which periodically moves data from the hot storage of the in-memory partitions into the cold storage. The decision is based on a flag in a dedicated column of the hybrid table.

Configuration Scenarios

In order not to interfere with the memory management of the in-memory engine, the IQ engine is usually deployed at a separate host/cluster. This allows for a more specific sizing of the involved nodes in the SAP HANA distributed landscape. For example, the extended storage may rely on a more powerful I/O subsystem than the server where the SAP HANA database is running and usually requires less main memory. The notion of a data platform ensures that the overall system landscape is kept manageable because of a unified installer and integrated administration tool chain.

Transactions

From an application perspective, both extended tables and hybrid tables appear like regular row or column tables in SAP HANA. This implies that they can participate in update transactions that insert, update or delete data in the extended or hybrid table. As a consequence, database operations on SAP HANA extended tables participate in (normal) distributed HANA transactions. In such a scenario, SAP HANA coordinates the transaction, e.g. generating the transaction IDs and commit IDs to integrate extended storage. As a seamless integration, we use the improved two-phase commit protocol described in [14] also for the extended storage. Consequently, SAP HANA will be able to recover the data in case of failures as any other SAP HANA table, including point-in-time recovery. In case of an error of the extended system, every access to a SAP HANA table may throw a runtime error. In particular, any query that touches objects located on the extended storage will be aborted. Additionally, if that access is part of a transaction that also touches in-memory column tables in SAP HANA, the entire transaction will be aborted. Finally, the recovery of an extended storage instance is recovered jointly with SAP HANA. Without this integrated recovery any transaction that had touched the extended store but not committed will be marked as "in-doubt". Clients will have the ability to manually abort these "in-doubt" transactions.

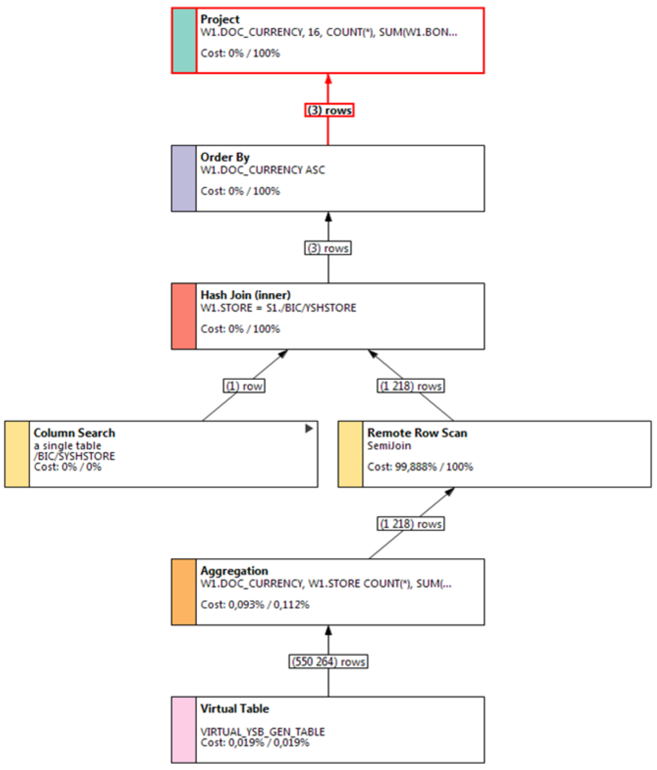


Figure 7: Federated query processing

Query Processing

Due to the tight integration of the IQ engine into the SAP HANA distributed query processing framework, the SAP HANA engine is able to exploit a huge variety of query processing tweaks, especially push-downs to the IQ storage manager. Capabilities include the ability to perform inserts, updates or deletes, order by, group by, different kinds of joins, or execution of nested queries. The cost-based query optimizer of SAP HANA either uses q-optimal histograms based on values for cardinality estimates on the extended storage [16]. The query optimizer considers communication costs for the data access to the extended storage. The *distributed exchange* operator is used to mark the boundary between HANA-local processing and remote execution in the IQ query processor. Sub plans below this operator are executed on the remote data source. During query optimization different alternatives exists to evaluate subplans in the extended storage:

- **Remote Scan:** Process a complete sub query independent from SAP HANA in the extended storage. The optimizer decides if the result should be returned in columnar or row-oriented format for further processing in SAP HANA.

- **Semijoin:** In this alternative data is passed from SAP HANA to the extended storage where it is used for filtering either in an IN-clause or a join using a temporary table. The optimizer picks such a strategy for example if parts of the fact table are sitting in IQ and require a join with smaller (and usually frequently updated) dimension tables.

- **Table Relocation:** This alternative considers SAP HANA tables as remote tables for IQ and pulls data on demand from SAP HANA.
- **Union Plan:** Do local processing in SAP HANA and the IQ storage engine, and use a union to integrate the partial results. Such a strategy is usually picked, if the dedicated aging flag allows for partition-local joins and partial group-bys.

Figure 7 shows the plan for a query that joins a columnar SAP HANA table with a selective local predicate with a table in the extended storage. In this scenario, the semijoin strategy is the most effective alternative because only a single row is passed from SAP HANA to the extended storage where it can be used to filter the large remote table. In this example it is even possible to push the group-by operation to the remote data source.

3.2 HANA Stream Extension

The HANA stream extension is based on the SAP Sybase Event Stream Processor (SAP Sybase ESP) and addresses the specific needs of data acquisition and responding with actions in realtime in a Big Data scenario. The data acquisition part deals, for example, with filtering out irrelevant events or immediate aggregation of the captured data for further processing in the SAP HANA core engine or Hadoop. The SAP Sybase ESP may also detect predefined patterns in the event stream and trigger corresponding actions on the application side.

To illustrate these use cases consider a telecom company that captures live data from its mobile network, for example information on the health status of its network or unexpected drops of customer calls (figure 8). In this figure, multiple components of the SAP HANA platform interact; each component is shown in a separate box with distinct color. In this telecom scenario, data is generated in high volume in the telecom network, but not all data has high value. Sensors in the telecom network capture various events, e.g. information about started or terminated phone calls. As long as the network is in a healthy state, no specific action has to be taken, and it is sufficient to store aggregated information on the status of the network. Hence, the SAP Sybase event stream processor uses the Continuous Computation Language (CCL)[2] to analyze the raw event data, instantly analyze it, and aggregate events over a predefined time window for further processing.

Furthermore, the raw data may be pushed into an existing HDFS using a dedicated adapter such that it is possible to perform a detailed offline analysis of the raw data. The resulting network data archive is then analyzed using map-reduce jobs in the Hadoop cluster. Such an advanced analysis might attempt to optimize the network utilization or to improve the energy efficiency of the network. In this scenario we use Hadoop for archiving purposes as the incoming data has highly variable structure. For more strictly structured data using the extended storage would be more appropriate.

In a development system the raw events collected in the network data archive may be replayed to the event stream processor to verify the effectiveness of improved event patterns. If the patterns derived through the map-reduce jobs in the Hadoop cluster prove useful, they are deployed in the productive ESP system.

However, if an outage of the network is detected, immediate action is required, which could be directly triggered by the SAP Sybase ESP and forwarded to the SAP HANA database using the SAP Data Services [2]. While an alert is immediately sent to the operations staff, a detailed analysis of the imported event data of

[2]CCL is a SQL-like declarative language to process events over windows of data, see [12] for details

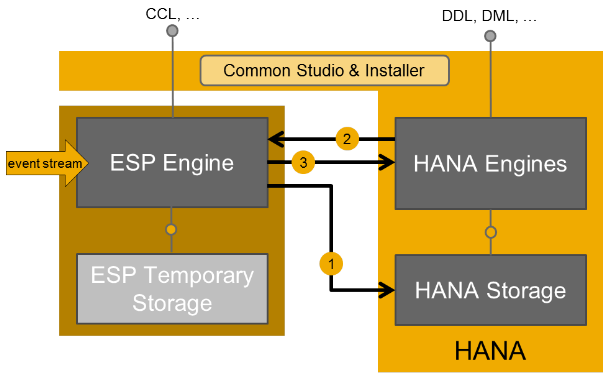


Figure 9: Example Scenario for Complex Event Processing in SQL HANA data platform

the exceptional event may also trigger reports to be prepared for service engineers. In a similar way, business data, e.g. information about the amount of transferred data but also standard business processes like billing or marketing are processed in the SAP HANA database.

Moreover, pre-aggregated and cleansed data may be loaded into the SAP HANA database for online processing. For example, the sensor data of antennas may be normalized into equi-distant measures of the network state and loaded into a time series table of SAP HANA. This allows advanced analysis on that data, e.g. perform correlation analysis between different sensors. Furthermore, the sensor data may trigger business actions because information about established connections collected from the sensor data must be matched with customer data for accounting purposes. This calls for an integrated processing runtime of sensor data and relational data stored in the SAP HANA database.

Use Cases

In general, the integrated ESP engine tackles the following three main use cases, also depicted in figure 9:

1. **Prefilter/pre-aggregate and forward:** In this use case, incoming data streams are filtered and/or aggregated. The result is then forwarded and directly stored within native SAP HANA tables. Although the ESP provides some temporary storage capability to reconstruct the content of a stream window, the forward use case allows to permanently store the window content under the control of the database system.
2. **ESP join:** In this case, slowly changing data is pushed during CCL query execution from the SAP HANA store into the ESP and there joined with raw data elements. For example, city names are attached to raw geo-spatial information coming from GPS sensors.
3. **HANA join:** In the opposite of a HANA join, a native HANA query may refer to the current state of an ESP window and use the content of this window as join partner within a relational query. Such a setup is particularly interesting, when dynamic content, e.g. the current state of a machine or environmental sensors, is required to be merged with traditional database content.

As common in stream environments, no transactional guarantees are provided; This however is usually accepted in high-velocity scenarios.

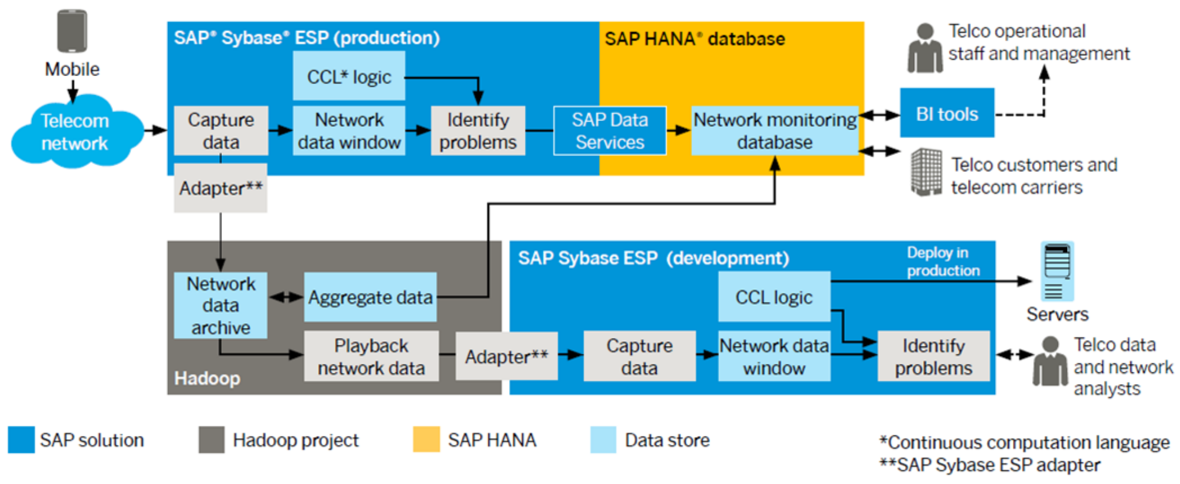


Figure 8: Example Scenario for Complex Event Processing in SQL HANA data platform

3.3 Summary

The overall strategy of SAP to rely on a native SAP HANA Big Data story as well as embracing (usually existing) open source infrastructure components requires to provide the concept of a data platform with an integrated set of different components. In this section we outlined the native extension points of SAP HANA with respect to IQ as the extended storage as well as ESP as the integrated stream engine allowing cross querying between the table-based database and the stream-based event processing world.

4. HANA INTEGRATION POINTS WITH HADOOP

Huge amounts of data are already stored in HDFS, and significant investments were made to analyze the data stored in the HDFS using either custom-made map-reduce jobs or using more declarative languages like HiveQL [25]. Clearly, for SAP customers want to tap into this ocean of data and generate higher-value information from it using these map-reduce jobs or Hive. In many cases, the resulting information must be integrated with the enterprise data warehouse or operational applications, e.g. for a detailed customer analysis.

In this section, we explain the goals we want to achieve with the Hadoop integration based on some customer scenarios. After that we survey technical details of the Smart Data Access (SDA) framework which is used to integrate SAP HANA with Hadoop. We focus on the Hadoop-side caching which results in significantly lower response times of map-reduce jobs and more effective use of the Hadoop cluster.

The integration of SAP HANA with any Hadoop distribution illustrates how easy and powerful it is possible to link almost any external data source with SAP HANA in a loosely coupled way. This open platform strategy is mainly realized with the Smart Data Access (SDA) technology, SAP HANA's capability-based adapter framework.

4.1 Goals and scenarios

Within the previous section we already pointed out that SAP ESP may push raw events immediately into the Hadoop Distributed File System (HDFS) so that it can be analyzed further in a rather offline fashion. In addition, significant amounts of data today are already stored into the HDFS. Usually, the shear amount of data (volume) but also its loose structure (high variety) makes it unattractive to load this data immediately into a relational database. Since raw data is of low value, freshness is not critical, and it is often sufficient to thoroughly analyze this data in batch jobs and then exposing the results to the SAP HANA database for reporting, planning etc.

In general, such a setup implies a co-existence of a conventional database side by side with data stored in HDFS which is analyzed in Hadoop [26, 17, 24, 6]. Map-reduce jobs are periodically executed to derive higher-level information in a more structured form which may be stored in a data warehouse but also to correlate the content stored in a HDFS with data already available in a database.

In this loosely coupled setup two main scenarios exist to link the database with Hadoop:

- Delegation of ad-hoc queries to Hadoop using an ODBC adapter and Hive [25].
- Exposing the content in the HDFS by calling existing map-reduce programs.

Having SAP HANA as the federation platform with Hadoop then leads to a setup with two separate systems. Standard tools for HANA, e.g. the eclipse-based administration and development tool SAP HANA Studio, can be used to develop Big Data applications. With these development tools at hand, SAP HANA can also handle life cycle tasks, e.g. transporting map-reduce programs from a development system to the productive system. As data now resides both in SAP HANA row/column in-memory tables and HDFS, query processing is distributed over both SAP HANA and Hadoop. In such a setup, SAP HANA serves as the main interface for ad-hoc queries and orchestrates federated query processing between these stores. If Hive is used, parts of a query may even be shipped to Hive based on the capabilities registered for Hive and Hadoop. Hadoop returns its result in a structured form that is ready to be consumed by HANA for further processing.

Having an interface from SAP HANA to Hadoop exposes several features of Hadoop which are typically not available in a relational database. For example, one can reuse libraries implemented on top of Hadoop like Mahout for machine learning, or custom social media analysis [11]. These kinds of tasks are typically a weak side of relational databases and can be softened using a Hadoop infrastructure. Moreover, one can use Hadoop as a scalable and very flexible way to implement user-defined functions which are capable to access schema-flexible data without the need to transform them into the relational model beforehand. Having SAP HANA as federation layer also allows us to combine relational query processing and Hadoop with the capabilities of statistical analysis with R [10].

To illustrate these use cases consider a project with a large SAP customer in the automotive industry with the objective to predict warranty claims. The data sources stored in SAP HANA on the one side included condensed information on the production, assembly and sales of automobiles but also facts about customers and past marketing campaigns. The raw data stored in HDFS on the other side included diagnosis read-outs on cars, support escalations, warranty claims and customer satisfaction surveys. Overall, the Hadoop cluster used twenty servers with 250 CPU cores, 1500 GB RAM and 400 TB Storage as aggregated compute power available in the Hadoop cluster. The HANA server was equipped with 40 cores, 512 GB RAM and 2 TB disk storage. Using Hive, we extracted data from twelve months data for a specific car series and made it available to the SAP HANA database server. With the SAP predictive analysis library using the apriory algorithm thousands of association rules were discovered with confidence between 80% and 100%. The derived models then were used to classify new readouts as warranty candidates in real-time in the SAP HANA database.

4.2 SDA-based integration - Query shipping

An important building block for the integration of Hadoop with SAP HANA is the Smart Data Access (SDA) framework implementing an access layer for a wide variety of different remote data sources like Hadoop, Teradata or other relational and non-relational systems. Thereby, SAP HANA realizes an open strategy to integrate and federate remote data sources.

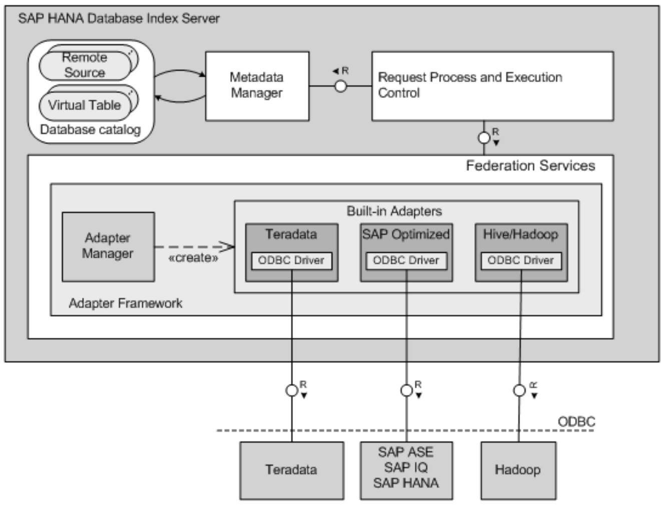


Figure 10: SAP HANA SDA Overview

Figure 10 gives a high-level overview of the generic SDA framework. In the SDA framework, remote resources are exposed as a *virtual table* to SAP HANA. Consequently, these resources can be referenced like tables or views in SAP HANA queries or views. The communication to remote resources is realized by adapters which are usually specific to the data source. There are various SDA adapters already available, e.g. Hadoop[3], any SAP database, IBM DB2, Oracle, or Teradata. This means that by providing an SDA adapter for a specific type data source makes this data source accessible for SAP HANA. As each SDA-adapter exposes capabilities specific to this data source, the SAP HANA query optimizers is able to forward parts of a query execution plan to the remote data source. In the remainder of this section we focus on the integration of Hadoop using SDA adapters as one prominent example, but most concepts also apply to other data sources.

[3]With HANA SPS09 we support the Hadoop distributions Intel IDH, Hortonworks, and Apache Spark.

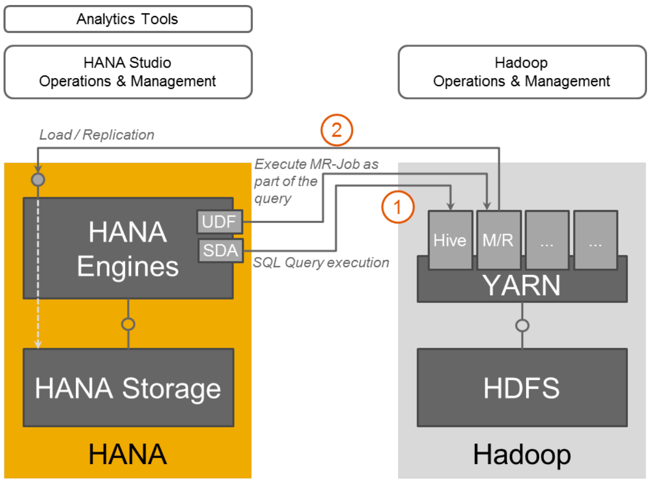


Figure 11: SAP HANA / Hadoop side-by-side

Registering a Remote Data Source

We use SDA to communicate with Hadoop via Hive, especially to address ad-hoc data analysis on data stored in HDFS. As indicated in figure 10, we use ODBC to establish the connection to the Hadoop system. In this setup SDA passes partial or complete queries or DDL statements to Hive for execution in Hadoop. The SDA framework and its integration into the HANA query compiler takes care that only queries are passed to Hadoop that are also supported by Hive and Hadoop. SDA also applies the required data type conversions with Hadoop. Below we show a typical workflow to first create the remote access to a Hive-based Hadoop distribution, wrap the remote source as a virtual table, and finally query the content of the data:

```
CREATE REMOTE SOURCE HIVE1 ADAPTER "hiveodbc"
       CONFIGURATION 'DSN=hive1'
 WITH CREDENTIAL TYPE 'PASSWORD' USING
       'user=dfuser;password=dfpass';
CREATE VIRTUAL TABLE "VIRTUAL_PRODUCT"
       AT "HIVE1"."dflo"."dflo"."product";
SELECT product_name, brand_name
  FROM "VIRTUAL_PRODUCT";
```

Query Processing

As mentioned above, SDA relies on a description of the capabilities of a remote server. For example, transactions for some database servers e.g. updates and transactions, are supported. However, for Hive and Hadoop only select statements without transactional guarantees are supported. For Hive on Hadoop it is possible, e.g. to push predicates or joins to Hadoop, but also to use semi-join reduction for faster distributed joins between Hadoop data and HANA tables. In the capability property file one finds, e.g. `CAP_JOINS : true` and `CAP_JOINS_OUTER : true`, to denote that inner joins and outer joins are supported. It is even possible that complete queries are processed via Hive and Hadoop. When only parts of a query can be executed in Hive and Hadoop, the query compiler generates a transient virtual table that represents the result of the subquery which will be processed by Hive and Hadoop. It also adds the needed operations to integrate this subquery into the plan executed in SAP HANA. In the simple-most case the results are only integrated via joins or union, but in more complex cases compensating operations might be required, e.g. mapping of data formats and data types.

To estimate the costs for accessing the Hadoop data, we rely on the statistics available in the Hive MetaStore, e.g. the row count and number of files used for a table. These statistics and also estimated communication costs are considered for generating the optimal federated execution plan. The plan generator attempts to minimize both the amount of transferred data and the response time of the query.

4.3 HANA Direct Access to HDFS

Besides the ad-hoc query capabilities via Hive discussed above, SAP HANA can also invoke custom map-reduce in Hadoop, see figure 11. This allows customers to reuse their existing map-reduce codebase and to access the HDFS files directly, i.e. without the additional Hive layer. The basic workflow is shown below: The first statement registers an Hadoop cluster with SAP HANA. The subsequent statement declares a virtual remote function that exposes an existing map-reduce job as a regular table function in SAP HANA. Finally, this virtual table function can be used in SQL statements like any other table function in SAP HANA.

```
CREATE REMOTE SOURCE MRSERVER
    ADAPTER hadoop CONFIGURATION
      'webhdfs=http://mrserver1:50070;
       webhcatalog=http://mrserver1:50111'
       WITH CREDENTIAL TYPE 'password'
       USING 'user=hadoop;password=hadooppw';

CREATE VIRTUAL FUNCTION
       PLANT100_SENSOR_RECORDS( )
    RETURNS TABLE (EQUIP_ID VARCHAR(30),
                   PRESSURE DOUBLE)
    CONFIGURATION
      'hana.mapred.driver.class =
    com.customer.hadoop.SensorMRDriver;
     hana.mapred.jobFiles =
              job.jar, library.jar;
     mapred.reducer.count = 1'
     AT MRSERVER;

SELECT A.EQUIP_ID, A.LAST_SERVICE_DATE,
       B.PRESSURE
FROM   EQUIPEMENTS A  JOIN
       PLANT100_SENSOR_RECORDS() B
       ON A.EQUIP_ID = B.EQUIP_ID
WHERE  B.PRESSURE > 90;
```

It is worth noting that this workflow is also embedded into SAP HANA's development environment and landscape management. For example, views accessing the map-reduce job as a table function keep their connection when the models are transported from test to productive system.

4.4 Remote Materialization

The map-reduce jobs in Hadoop are typically used to turn large volumes of rather low-value data into smaller data sets of higher value data. As the runtime of these map-reduce jobs tends to be significant, SAP HANA offers to cache the result of map-reduce jobs on Hive side. As customers have a sound understanding of their business data they will know where absolute data freshness is not needed, e.g. for low velocity data. The freshness of the cached results of map-reduce jobs is configurable.

Extended Query Processing

When an application submits a query that includes a Hive table as a data source, it can request the use of the caching mechanism. More precisely, the application appends `WITH HINT (USE_REMOTE_CACHE)` to the query string to enable the cache usage. During query processing it is checked if caching is requested for the query. If this is the case, a hash key is computed from the HiveQL statement, parameters, and the host information. With this hash key we can can ensure that the same query is cached at most once. If a value is cached for this hash-key the corresponding query result is returned immediately. If either no caching is requested, no cache key was found or the cached value is older than a configurable parameter, the query is evaluated from scratch. Evidently, the cached data is not related to specific transactional snapshots, but we expect this to be of little concern in a Hive setup. The cached data is stored in temporary tables in HDFS. Large caches are possible because the HDFS often has enough space to keep materialized query results in addition to the raw data. Overall, caching leads to a more effective use of the compute resources in the Hadoop cluster.

To illustrate these concepts, consider the following query on the TPC-H schema which references two virtual tables `CUSTOMER` and `ORDERS` from Hive:

```
SELECT c_custkey, c_name,
       o_orderkey, o_orderstatus
FROM   customer JOIN orders
       ON c_custkey = o_custkey
WHERE  c_mktsegment = 'HOUSEHOLD'
```

Under normal execution, i.e. without any caching in Hive, when the federated query is executed at the remote source, the Hive compiler generates a DAG of map-reduce jobs corresponding to the federated query. Map-reduce jobs from this DAG are triggered thereafter, upon completion of which the results are fetched back into HANA for further operations. The query execution plan for the normal execution mode of the example query is shown in figure 12. It shows two Virtual Table nodes corresponding to the tables `CUSTOMER` and `ORDERS` with their corresponding predicates, and a `Nested Loop Join` node which is also computed at the remote source. Since, there are no other tables accessed locally inside SAP HANA, the data received in the `Remote Row Scan` node is projected out. If the query has references to other local HANA tables, the data received in the `Remote Row Scan` node will be used for further processing along with data from the local tables.

Under the enhanced mode with remote caching enabled, the optimizer identifies the hint to use remote materialization and materializes the results, after executing the DAG of map-reduce jobs generated by the Hive compiler, to a temporary table at the remote site. It then modifies the query tree to read everything from this temporary table. It should be noted that this materialization process is a single-time activity and every subsequent execution of this federated query will fetch the results from the materialized copy stored in the temporary table instead of executing the corresponding DAG of map-reduce jobs. The modified query execution plan for our example query under the enhanced mode is shown in figure 13. It shows one `Virtual Table` node from which all the interesting

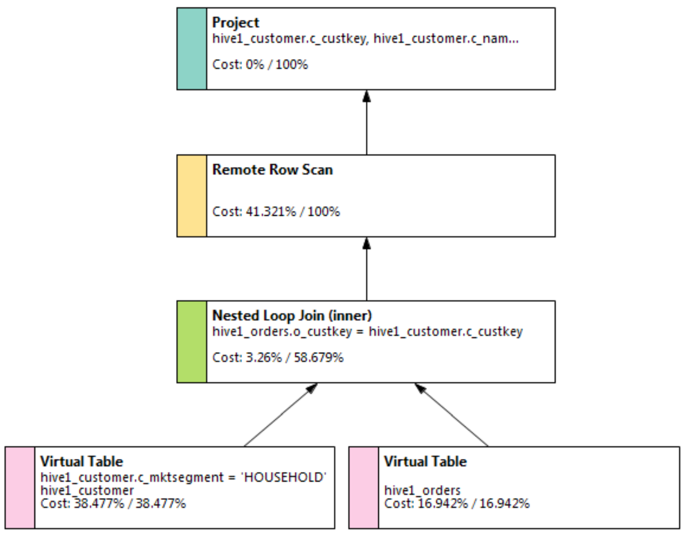


Figure 12: Query plan without remote materialization

data required to answer the query can be retrieved. For the current example, this virtual table node corresponds to the joined data from the tables `CUSTOMER` and `ORDERS` with all the necessary predicates already applied. We can see from the execution plan in figure 13 that the `Remote Row Scan` node is directly fetching the necessary data from the temporary table, with no additional predicates being applied on the temporary able.

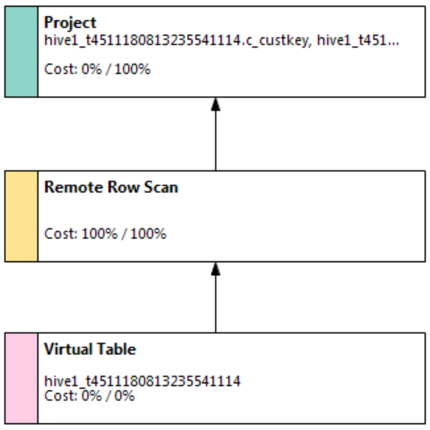


Figure 13: Query plan with remote materialization

In addition to these basic caching techniques the remote materialization implements further improvements: First, we only materialize queries with predicates. This ensures that we do not replicate the entire Hive table as a materialized result set as this will not add any value to the performance of the system. Second, the duration for which a materialized result set is valid and is persisted in the remote source is controlled via a configuration parameter, called `remote_cache_validity`. When the query optimizer identifies the hint `USE_REMOTE_CACHE`, it checks if the materialized data set on the remote source is valid based on this parameter setting before actually using it. If it discovers that the data set is outdated, it discards the old data set and materializes the result set to a new copy at the remote source. Finally, the remote materialization enhancement in SAP HANA is disabled by

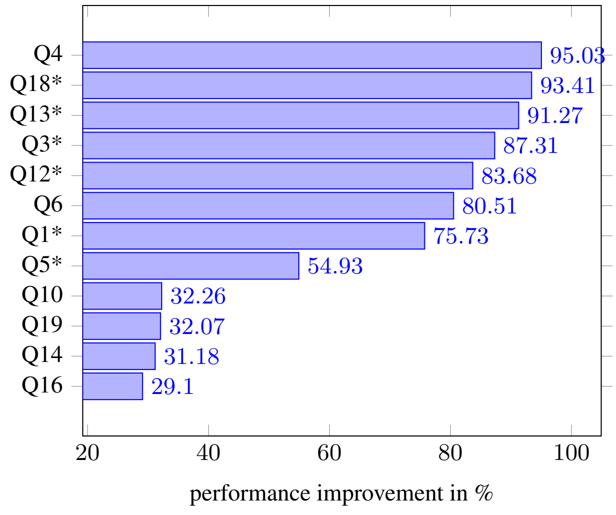


Figure 14: Runtime benefit of remote materialization

default and can be controlled using the configuration parameter `enable_remote_cache`. This parameter is useful in scenarios when the customer needs to completely disable the feature, for example, low storage availability in HDFS or when the tables in Hive are being frequently updated.

Performance Analysis

We demonstrate the difference between using Hive and SAP HANA with and without remote materialization. This small experiment uses the TPC-H dataset with scale factor 1. The experiments were performed using SAP HANA SPS07 as the primary database running SUSE SLES 11.2 on a server with 16 physical cores and 256GB RAM. The 7-node Hadoop cluster was accessed from SAP HANA via Hive's ODBC driver as remote source. We used a Hadoop cluster configuration with Apache Hadoop 1.0.3, Hive 0.9.0 on an HDFS with 21.5TB capacity, 240 map tasks, 120 reduce tasks, and 6 worker nodes. The following tables from the TPC-H schema were federated remotely at Hive: `LINEITEM`, `CUSTOMER`, `ORDERS`, `PARTSUPP`, and `PART`. The tables present locally in SAP HANA were: `SUPPLIER`, `NATION`, `REGION` (, and `PART` only for Q14 and Q19). Such a small scale-factor is ridiculously small for a typical Hive and Hadoop setup, but for large data sets the positive impact of remote materialization would be even more pronounced. In that sense, this somewhat unrealistic setup is a very conservative analysis of the expected performance improvements of remote materialization with SAP HANA and Hive.

We used slightly modified versions of the benchmark queries. In particular, we removed the `TOP` and `ORDER BY` clauses from the TPC-H queries, with the exceptions being those queries for which the sorting was done inside SAP HANA. This is desirable as we cannot make any assumptions about the ordering property of the datasets fetched from Hive which were materialized earlier.

In figure 14 we present the runtime benefit achieved when using SDA with remote materialization enhancement using the normal execution mode with SDA as the baseline. We also demonstrate the materialization overhead incurred in materializing the results on the remote system; this is shown in figure 15. We have marked the modified queries discussed above with an asterisk (*).

We can see from figure 14 that using remote materialization, some queries can benefit as high as 95% with respect to query re-

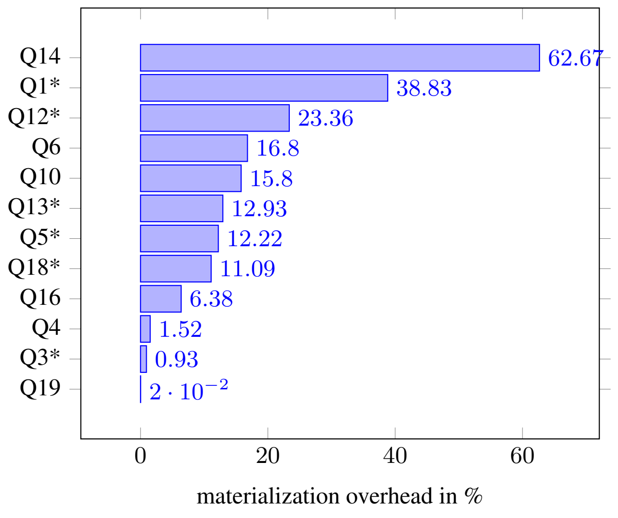


Figure 15: Materialization overhead of remote materialization

sponse time. There are two aspects to the overall query execution time when we have a remote system in tandem with SAP HANA: time taken to fetch the data required from the remote source, and time taken to join the fetched data with local data in SAP HANA. We can infer from the results that the queries can be divided into two segments based on their respective performance gains.

The data displayed in figure 14 has been sorted based on the maximum runtime benefit. The top seven queries for scale factor 1 demonstrate high gain of more than 75%. This is expected because all the tables accessed in these queries are federated tables, and there are no local tables with which the results fetched from the remote source are joined. For the remaining queries, the performance gain is on the lower side because the results fetched from the remote source are joined with local tables in HANA. This is also expected since we demonstrate the percentage improvement in the overall query execution time, and not just the time taken to fetch the data required from the remote source. We can conclude from the results of our experiment that our enhancement provides maximum benefits for the cases when the majority of the computation is performed on the remote system and minimum data is read back into HANA. In an analytical workload, several queries are executed multiple times and with remote materialization enhancement, every execution of the query can benefit by skipping this computation and directly fetching the already materialized data. This enhancement is specifically useful in case of Hive, because a user may not have exclusive access to the Hadoop cluster and may only get a limited share of the Hadoop cluster's overall capacity.

The materialization process at the remote source has an associated overhead with it, which is also demonstrated in figure 14. This overhead is the additional time required to materialize the results in Hive and is a single-time cost, which is incurred when the query is first executed with the optimizer hint `USE_REMOTE_CACHE`. As long as the data in the Hive tables is not modified, SAP HANA can be configured to continue reusing the materialized results from Hive. This can have huge benefits with low velocity data stored in Hive, depending on the frequency of queries that use these materialized results. This overhead in materialization process is partly because `CREATE TABLE AS SELECT` (or `CTAS`) in Hive is currently a two-phase implementation: first the schema resulting from the `SELECT` part is created, and then the target table is created. We make use of this `CTAS` infrastructure provided by Hive to create the temporary tables. The materialization overhead demonstrated in our experiments is set to go down when the `CTAS` implementation in Hive gets optimized.

5. SUMMARY

The SAP HANA data platform reflects SAP's answer to the ever-increasing requirements and opportunities in management of data. It was specifically designed as a single point of access for application logic by providing a HANA native as well as HANA open strategy:

- SAP HANA core database can serve real time, complex queries and multi-structured data needs.
- SAP Sybase IQ (HANA IQ) can provide highly concurrent OLAP workload in combination with large scale, disk-based storage
- SAP Sybase ESP (HANA ESP) can provide high velocity on-the-fly analysis with native hand-over to other SAP HANA components.
- Hadoop can provide cheap data storage and pre-processing for large scale unstructured/unformatted data and compiled into the HANA data platform by allowing query/code push-down and part of HANA's life cycle management capabilities.
- SAP BW and SAP EIM [2] – as some examples on an application level – can provide consumption, modeling, and integration capabilities (including Hadoop)

This paper gives some insight into the overall picture of Big Data applications in SAP's perception as well as diving into technical details of the HANA native and HANA open integration strategy.

Acknowledgment

We would like to thank all colleagues working on the Big Data strategy and productization of SAP HANA. We especially thank the team of Sudipto Chowdhuri which did the main part of the implementation presented in this paper.

6. REFERENCES

[1] *Delivering Continuity and Extreme Capacity with the IBM DB2 pureScale Feature*. `http://www.redbooks.ibm.com/abstracts/sg248018.html`.
[2] *SAP Enterprise Information Management*. `http://help.sap.com/eim`.
[3] *SAP Replication Server Documentation*. `http://help.sap.com/replication-server`.
[4] *SAP SQL and System Views Reference*. `http://help.sap.com/hana_platform/`.
[5] S. Chaudhuri, U. Dayal, and V. Narasayya. An overview of business intelligence technology. *Commun. ACM*, 54(8):88–98, Aug. 2011.
[6] D. J. DeWitt, A. Halverson, R. V. Nehme, S. Shankar, J. Aguilar-Saborit, A. Avanes, M. Flasza, and J. Gramling. Split query processing in Polybase. In *SIGMOD Conference*, pages 1255–1266, 2013.
[7] C. Diaconu et al. Hekaton: SQL Server's memory-optimized OLTP engine. In *SIGMOD*, pages 1243–1254, 2013.

[8] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA Database – an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[9] G. Graefe. The five-minute rule 20 years later: And how flash memory changes the rules. *ACM Queue*, 6(4):40–52, July 2008.

[10] P. Große, W. Lehner, T. Weichert, F. Färber, and W.-S. Li. Bridging two worlds with RICE – integrating R into the SAP in-memory computing engine. *Proc. VLDB*, 4(12):1307–1317, 2011.

[11] P. Große, N. May, and W. Lehner. A study of partitioning and parallel udf execution with the SAP HANA database. In *SSDBM*, page 36, 2014.

[12] S. Inc. *Stream, Schema, Adapter, and Parameter CCL Language Extensions*. `http://www.sybase.de/detail?id=1080119`.

[13] P.-Å. Larson, E. N. Hanson, and S. L. Price. Columnar storage in SQL Server 2012. *IEEE Data Eng. Bull.*, 35(1):15–20, 2012.

[14] J. Lee, Y. S. Kown, F. Färber, M. Muehle, C. Lee, C. Bensberg, J. Y. Lee, A. H. Lee, and W. Lehner. SAP HANA distributed in-memory database system: Transaction, session, and metadata management. In *ICDE*, 2013.

[15] R. MacNicol and B. French. Sybase IQ multiplex - designed for analytics. In *VLDB*, pages 1227–1230, 2004.

[16] G. Moerkotte, D. DeHaan, N. May, A. Nica, and A. Boehm. Exploiting ordered dictionaries to efficiently construct histograms with q-error guarantees in SAP HANA. In *SIGMOD*, pages 361–372, 2014.

[17] F. Özcan, D. Hoa, K. S. Beyer, A. Balmin, C. J. Liu, and Y. Li. Emerging trends in the enterprise data analytics: Connecting Hadoop and DB2 warehouse. In *SIGMOD*, pages 1161–1164, 2011.

[18] J. Prabhala. The future is now for SAP HANA, last accessed Jan. 17th 2015, June 2014. `http://www.advizex.com/blog/future-now-sap-hana/`.

[19] I. Psaroudakis, F. Wolf, N. May, T. Neumann, A. Boehm, A. Ailamaki, and K.-U. Sattler. Scaling up mixed workloads: a battle of data freshness, flexibility, and scheduling. In *TPCTC*, page accepted for publication, 2014.

[20] V. Raman et al. DB2 with BLU acceleration: So much more than just a column store. *Proc. VLDB Endow.*, 6(11):1080–1091, Aug. 2013.

[21] G. W. Records. Largest data warehouse, last accessed Jan. 17th 2015. `http://www.guinnessworldrecords.com/world-records/5000/largest-data-warehouse`.

[22] M. Rudolf, M. Paradies, C. Bornhövd, and W. Lehner. The graph story of the SAP HANA database. In *BTW*, pages 403–420, 2013.

[23] R. Stoica, J. J. Levandoski, and P.-Å. Larson. Identifying hot and cold data in main-memory databases. In *ICDE*, pages 26–37, 2013.

[24] X. Su and G. Swart. Oracle in-database Hadoop: When mapreduce meets RDBMS. In *SIGMOD*, pages 779–790, 2012.

[25] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, Aug. 2009.

[26] Y. Xu, P. Kostamaa, and L. Gao. Integrating Hadoop and parallel DBMs. In *SIGMOD*, pages 969–974, 2010.

Taxi Queue, Passenger Queue or No Queue?

A Queue Detection and Analysis System using Taxi State Transition

Yu Lu, Shili Xiang, Wei Wu
Institute for Infocomm Research, A*STAR, Singapore
{luyu, sxiang, wwu}@i2r.a-star.edu.sg

ABSTRACT

Taxi waiting queues or passenger waiting queues usually reflect the imbalance between taxi supply and demand, which consequently decrease a city's traffic system productivity and commuters' satisfaction. In this paper, we present a queue detection and analysis system to conduct analytics on both taxi and passenger queues. The system utilizes the event-driven taxi traces and the taxi state transition knowledge to detect queue locations at a coordinate level and subsequently identify 4 different types of queue context (e.g., only passengers queuing or only taxis queuing). More specifically, it adopts the novel and easy-to-implement algorithms to selectively extract taxi pickup events and their critical features. The extracted taxi pickup locations are then used to detect queue locations, and the extracted critical features are used to infer queue context. The extensive empirical evaluations, which run on daily 12.4 million taxi trace records from nearly 15000 taxis in Singapore, demonstrate the high accuracy and stability of the queue analytics results. Finally, we discuss the real world deployment issues and the gained insights from the queue analysis results.

1. INTRODUCTION

In the densely populated Asian cities (e.g., Singapore, Beijing and Taipei), relatively cheap taxi fares and large number of taxis greatly facilitate the pervasive usage of taxis by urban citizens for various purposes, such as traveling between office and home, purchasing groceries at supermarkets and visiting friends. It is relatively different from the taxi usage at many cities in US or Europe, where taxis more frequently serve airport routes and do not cover all urban districts. The taxi usage characteristics in the Asian cities easily cause that the temporal and spatial imbalance of taxi supply and demand occurs frequently: taxis would queue up for passengers due to temporarily low taxi demand but high supply nearby; passengers would queue up for taxis due to temporarily high taxi demand but low supply; in many time periods, taxis and passengers would concurrently queue up as both taxi demand and supply are high. Such queuing events usually not only reduce the productivity of an urban traffic system, but also greatly decrease the satisfaction of public commuters as well as taxi drivers. Fig. 1 illustrates a taxi queue and a passenger queue that both frequently occur in Singapore.

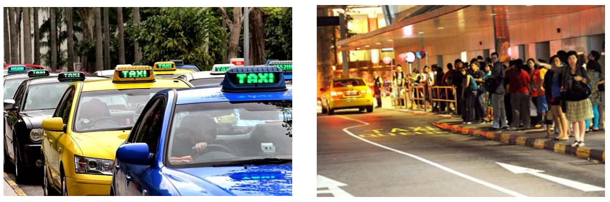

(a) Taxi Queue (b) Passenger Queue

Figure 1: Different Types of Queue in Singapore

Properly and accurately detection of queue locations and queue context would benefit many parties and stakeholders. The real time queuing events information and their long-term patterns can be used in the recommendation systems for taxi drivers and commuters (e.g., suggest commuters to the nearby taxi queue locations). The information can also be used in the taxi operators' booking and dispatching systems (e.g., guide available taxis to passenger queue locations). Moreover, the government agencies need such information to understand the imbalance between taxi supply and demand, and accordingly take necessary actions (e.g., increase operating taxis or adjust taxi fares).

Motivated by the availability of abundant information in taxi traces, e.g., GPS locations and taxi states, using taxi traces to design and build a city scale queue detection and analysis system is a promising solution. However, it is an open and non-trivial problem. Firstly, taxi queuing for passengers is not simply a passenger pickup, dropoff or vehicle parking event, only the GPS coordinates and the binary taxi states (occupied or non-occupied) are not enough to capture it. Secondly, passenger queuing for taxis is even more difficult to detect, as no any direct information from the passenger side and no apparent clue in taxi traces. Thirdly, both taxi queuing and passenger queuing are highly dynamic in terms of time and locations, which not only repeatedly occur at fixed taxi stands or during peak hours.

In this paper, we present a practical system that captures

10.5441/002/edbt.2015.60

taxis and passengers queuing activities at a fine-grained scale, i.e., at the individual coordinate level rather than a region or zone, and subsequently analyzes different types of queue context. We summarize the key contributions of this work as follows:

- We propose a novel approach, using multiple taxi states and their transition information, to conduct the city scale queue analytics for both taxis and passengers.
- We design and implement a two-tier queue analytics engine, where the lower tier module detects queuing locations and the upper tier module identifies queue context based on the selected taxi pickup events and features.
- We conduct the extensive empirical evaluation of our queue analytics results before we deploy the system in the real world. We demonstrate its stability using large scale taxi traces, and its accuracy using various other data sources (e.g., landmark information, failed taxi booking data).

The rest of the paper is organized as follows: section 2 introduces the background of the dataset, and then section 3 depicts our overall system architecture and define the queue types. In sections 4 and 5, we describe our queue spot detection and queue context disambiguation modules in detail respectively. Extensive empirical evaluations are conducted in section 6, which is followed by a discussion on deployment and other issues in section 6. The related work is presented in section 8. At last, we conclude with future work in section 9.

2. TAXI STATE AND EVENT-DRIVEN LOG

2.1 Mobile Data Terminal

As part of the taxi operators' efforts on improving their quality of service, each taxi in Singapore is equipped with a specifically designed device, called mobile data terminal (MDT), which is mainly used to handle taxi bookings and monitor a taxi's real time status. More specifically, it receives taxi booking tasks from the backend service (taxi call center), and sends back taxi driver's decision (accept or reject the task) via general packet radio service (GPRS). Moreover, MDT keeps logging and updating a taxi's real time state by collecting the information from taxi meter, roof-top signs and its frontend touch screen. Fig. 2 simply depicts an MDT system on a Singapore taxi, where MDT is hardwired directly to different on-vehicle devices and provides taxi drivers a multifunctional touch screen.

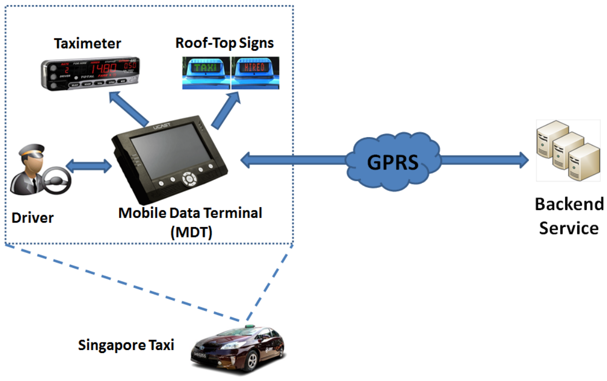


Figure 2: A simplified telematics system on a Singapore Taxi

2.2 Taxi State

Based on the collected real time information, the MDT device is able to precisely identify 11 different taxi states. Table 1 lists all the taxi states with their descriptions. The taxi state transitions mainly depend on the type of a taxi job. In principle, all taxi jobs can be classified into two categories: *street* job and *booking* job.

A *street* job means a taxi picks up new passengers by street hail, and the following is the typical taxi state transitions on a street job:

a) a passenger hails down a taxi with *FREE* state along a road or a taxi stand.
b) the taxi driver starts the taximeter for a new trip, and meanwhile the MDT updates the taxi state to *POB*.
c) during the trip, the taxi state keeps *POB* while the MDT periodically updates the taxi GPS location.
d) the taxi is approaching the destination and the driver presses the STC button on the MDT touch screen to update the taxi state to *STC*.
e) upon arrival of the destination, the driver presses the button on the taximeter for printing the receipt, and meanwhile the MDT updates the taxi state to *PAYMENT*.
f) once the driver resets the taximeter after the passenger alights, the MDT automatically updates the taxi state to *FREE* again.

A *booking* job means a taxi picks up new passengers, who have made a booking via telephone, short message service (SMS) or mobile phone applications (apps). The typical taxi state transitions on a booking job can be described as below:

a) a passenger makes a taxi booking, and the backend service dispatches the booking information to the nearby taxis with *FREE* or *STC* state.
b) a taxi driver successfully bids the booking job by pressing the button on the MDT touch screen, and meanwhile the MDT updates the taxi state to *ONCALL*.
c) upon arrival of the booking pickup location, the MDT updates the taxi state to *ARRIVED*.
d) if the passengers do not show up within a specific time period (e.g., 15 minutes), the MDT updates the taxi state to NOSHOW first and then to *FREE* within 10 seconds.
e) if the passenger gets on the taxi in time, the MDT updates the taxi state to *POB* once the driver starts the taximeter.
f) the subsequent taxi state transitions are the same as street job's procedure, i.e., from street job's step c) to step f).

Fig. 3 illustrates a complete taxi state transition diagram, which includes the procedures of both street jobs and booking jobs.

Table 1: Taxi State and Description

Taxi State	Description
FREE	Taxi unoccupied and ready for taking new passengers or bookings
POB	Passenger on board and taximeter running
STC	Taxi soon to clear the current job and ready for new bookings
PAYMENT	Passenger making payment and taximeter paused
ONCALL	Taxi unoccupied, but accepted a new booking job
ARRIVED	Taxi arrived at the booking pickup location and waiting for the passenger
NOSHOW	No passenger showing up and the booking canceled soon
BUSY	Taxi driver temporarily unavailable due to a personal reason
BREAK	Taxi on a break and driver logged on MDT
OFFLINE	Taxi on a break and driver logged off from MDT
POWEROFF	MDT shut down and not working

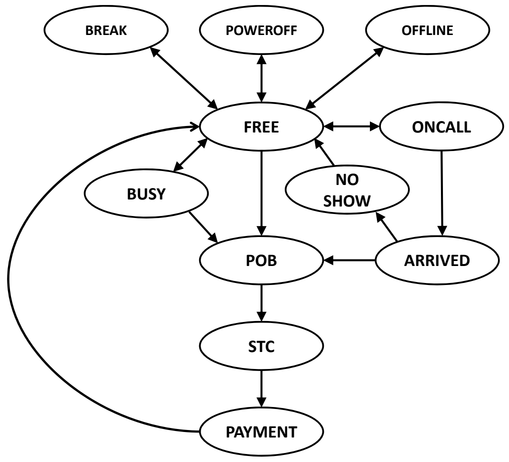


Figure 3: Taxi State Transition Diagram

2.3 MDT Log

As the central processing device on a taxi, MDT keeps updating and tracking any changes of taxi state and other critical information, e.g., GPS location, vehicle speed and taxi fares. The MDT logging module writes all such information to its local storage, and meanwhile selectively and periodically sends them to the backend service via GPRS. The MDT logging frequency is not fixed by default, and a logging action is triggered by the taxi state changes, GPS location updates and a few other critical vehicle events. Different from the traditional GPS localizer traces, the MDT log module adopts the event-driven logging mechanism, which are explicitly driven by the 11 taxi state transition events. Therefore, the MDT log captures much more accurate and abundant information than the traditional GPS traces, and accordingly provides more opportunities to discover and understand activities of both taxis and passengers.

We use the MDT log from a large local taxi operator, and select its 6 fields: timestamp, taxi ID, GPS location, instantaneous taxi speed and taxi state. Table 2 gives the selected fields in the MDT log and a sample record.

Table 2: Selected Fields of MDT Log with a Sample

Timestamp	Taxi ID	Longitude	Latitude	Speed	Taxi State
01/08/2008 19:04:51	SH0001A	103.7999	1.33795	54	POB

3. SYSTEM OVERVIEW

The system block diagram with the proposed Queue Analytic Engine is illustrated in Fig 4, and it mainly consists of two core modules:

- Queue Spot Detection Module: this module is the component for detecting queue locations (spots) based on the selected taxi pickup events. An algorithm is specifically designed for this module to extract pickup event sub-trajectories, which uses both the taxi state transition knowledge and the taxi instantaneous speed. In order to detect stable queue spots, it requires a relatively long-term historical dataset, e.g., a full day's MDT logs, from a large number of taxis.
- Queue Context Disambiguation Module: this module is the component for identifying different queuing situations, as defined in Table 3, at a queue spot. Based on the two newly proposed algorithms, the module firstly fetches the required features from the input of pickup event sub-trajectories, and then uses the features to resolve distinct queue types. The taxi state transition knowledge are used to accurately capture different time points in pickup events. This module mainly runs on a relatively short-term historical dataset for analyzing queue context transition patterns and queue types.

As mentioned earlier, Table 3 defines the four queue types, i.e., C_1 to C_4. The queue type C_1 is both taxi queue and passenger queue concurrently occur at the given queue spot, which indicates taxi demand and supply are presently both high. The queue type C_2 is only passenger queue, C_3 is only taxi queue, and C_4 is neither taxi queue nor passenger queue at the given queue spot.

In this paper, "queue" refers to a stable number of waiting entities during a specific time period, which indicates that the average arrival rate exceeds the average service rate. To

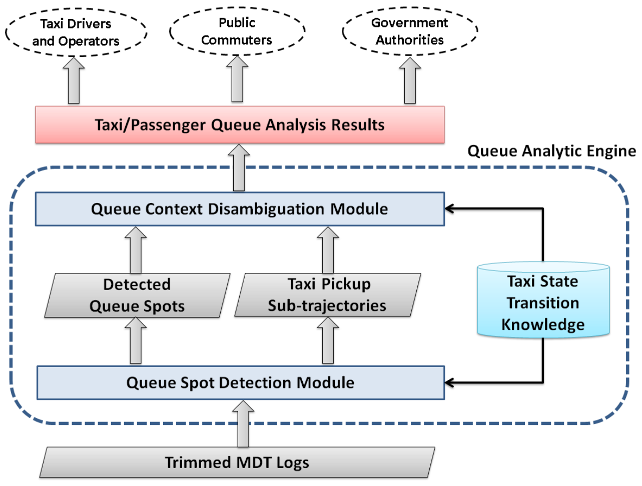

Taxi Drivers and Operators
Public Commuters
Government Authorities
Taxi/Passenger Queue Analysis Results
Queue Analytic Engine
Queue Context Disambiguation Module
Detected Queue Spots
Taxi Pickup Sub-trajectories
Taxi State Transition Knowledge
Queue Spot Detection Module
Trimmed MDT Logs

Figure 4: System Block Diagram of Queue Analytic Engine

Table 3: Four Types of Queue Context

Queue Type	Passenger Queue	No Passenger Queue
Taxi Queue	C_1	C_3
No Taxi Queue	C_2	C_4

clarify the above described queue types, we define taxi queue and passenger queue as below:

- Taxi Queue: One available taxi or more steadily awaiting for taking new passengers at a queue spot during a given time period.
- Passenger Queue: One passenger or more steadily awaiting for taxis at a queue spot during a given time period.

Note that no taxi queue or no passenger queue only means no stable waiting taxis or passengers, and it does not necessarily mean no any taxis waiting for a short time or passengers quickly getting taxis. Moreover, for both the taxi queue and the passenger queue, we do not impose any assumptions on the queue shapes and service modes, but only the first-come first-served (FIFO) discipline.

4. QUEUE SPOT DETECTION

It seems that queue spots can be easily detected by clustering the most frequent taxi pickup/dropoff locations and taxi parking locations. However, such a straightforward approach has difficulties due to two reasons. Firstly, a high proportion of *quick* pickup and dropoff events occur at any non-restricted locations in the city, and it would easily result the entire road rather than a small queuing area being a cluster. Secondly, a frequent taxi parking location is not necessarily a taxi queue spot for passengers, and thus the specific taxi state transitions need to be considered and checked.

We therefore only consider such pickup events: a taxi with an unoccupied state parks for a time of period and then departs with an occupied state. We propose a new algorithm to extract such slow pickup events, and we then determine taxi queue spots by clustering the most frequent locations of the extracted pickup events.

4.1 Preliminary

We firstly define and clarify several important terms which are used in the following sections.

Definition 1. Individual taxi's trajectory $\Re$: A temporally ordered sequence of the trimmed MDT log records from one taxi, i.e., $p_1 \rightarrow p_2 \rightarrow \cdots \rightarrow p_n$, where p_i $(1 \leq i \leq n)$ is the tuple containing the taxi state $p_{i.state}$, instantaneous speed $p_{i.speed}$, latitude coordinate $p_{i.lat}$, longitude coordinate $p_{i.lon}$ and timestamp $p_{i.ts}$.

Definition 2. Individual taxi's sub-trajectory $R(s, e)$: A segment of an individual taxi's trajectory, i.e., $p_s \rightarrow p_{s+1} \rightarrow \cdots \rightarrow p_e$, where $1 \leq s < e \leq n$.

Definition 3. Individual taxi's sub-trajectory set ω: A collection of an individual taxi's sub-trajectories, i.e., $\{R^k | k = 1, 2, \cdots\}$, where $R^k = R(s_k, e_k)$.

Definition 4. Multiple taxis' sub-trajectory set W: A collection of multiple taxi's sub-trajectory sets, i.e., $\{\omega^j | j = 1, 2, \cdots\}$, where ω^j is the j^{th} taxi's individual sub-trajectory set.

Based on the taxi state descriptions in Table 1, we classify the taxi states into three state sets:

Definition 5.1 Taxi occupied state set Θ: { POB, STC, PAYMENT }.

Definition 5.2 Taxi unoccupied state set Ψ: { FREE, ONCALL, ARRIVED, NOSHOW }.

Definition 5.3 Taxi non-operational state set Λ: { BREAK, OFFLINE, POWEROFF }.

The *BUSY* state is a special state, and we do not assign it into the above defined three taxi state sets. We will discuss it separately in the subsequent sections.

4.2 Pickup Event Extraction

In order to detect each individual taxi's slow pickup events, we propose a simple and practical algorithm, called pickup extraction algorithm (PEA): its input is an individual taxi's trajectory $\Re$ and output is the sub-trajectory set ω of the required taxi pickup events. The basic idea behind the PEA algorithm is that a slow taxi pickup event normally has at least two consecutive low speed records (e.g., below 10 km per hour) during the period of moving forward in the waiting line. Meanwhile, a pickup event shows certain taxi state transitions in the corresponding sub-trajectories, e.g., from *FREE* to *POB*. The complete algorithm is shown in Algorithm 1.

Algorithm 1 Pickup Extraction Algorithm

```
Input: A taxi's trajectory $\Re$ and speed threshold $\eta_{sp}$.
Output: The sub-trajectory set $\omega$.
1: $\delta_1 \leftarrow false$; $\delta_2 \leftarrow false$; $k \leftarrow 1$;
2: for $i = 1 \rightarrow n$ do
3:   if $p_{i.state} \notin \Lambda$ then
4:     if $p_{i.speed} \leq \eta_{sp}$ and $\delta_1$=false and $\delta_2$=false then
5:       $\delta_1 \leftarrow true$;
6:     else if $p_{i.speed} \leq \eta_{sp}$ and $\delta_1$=true and $\delta_2$=false then
7:       $R^k.Add(p_{i-1})$; $R^k.Add(p_i)$; $\delta_2 = true$;
8:     else if $p_{i.speed} \leq \eta_{sp}$ and $\delta_1$=true and $\delta_2$=true then
9:       $R^k.Add(p_i)$;
10:    else if $p_{i.speed} > \eta_{sp}$ and $\delta_1$=true and $\delta_2$=false then
11:      $\delta_1 \leftarrow false$;
12:    else if $p_{i.speed} > \eta_{sp}$ and $\delta_1$=true and $\delta_2$=true then
13:      if $p_{s_k.state} \in \Theta$ and $p_{e_k.state} \in \Psi$ then
14:        goto TAG1;
15:      else if $p_{s_k.state}$=FREE and $p_{e_k.state}$=ONCALL then
16:        goto TAG1;
17:      else if the taxi states in $R^k$ never change then
18:        goto TAG1;
19:      else
20:        $\omega.Add(R^k)$; $k \leftarrow k + 1$;
21:      end if
22:    else
23:      goto TAG1;
24:    end if
25:  else
26:    TAG1: $R^k \leftarrow \emptyset$; $\delta_1 \leftarrow false$; $\delta_2 \leftarrow false$;
27:  end if
28: end for
```

The proposed PEA algorithm firstly filters out the sub-trajectories with any of the non-operational taxi state, i.e., $p_{i.state} \notin \Lambda$. After that, it sets the low speed flag δ_1 to *true* when the first low speed is detected, i.e., the taxi speed falls below (or equal to) the speed threshold η_{sp}. Only when the subsequent taxi speed also falls below the speed threshold, it starts to add the tuple p_i into an empty sub-trajectory R^k and repeats such an adding action until the speed becomes bigger than the threshold again. Thus, all the extracted sub-trajectories have at least two tuples with the speeds below the given threshold.

Finally, PEA adds the new R^k into the sub-trajectory set ω, given R^k satisfies the following state transition constraints: 1) R^k does not start with an occupied state and end with an unoccupied state, i.e., $p_{s_k.state} \notin \Theta$ and $p_{e_k.state} \notin \Psi$, as it is simply a "passenger alight" event; 2) R^k does not start with FREE and end with ONCALL, as it means the taxi leaves for a new booking job at another location; 3) R^k has at least one-time state transition, as we need to filter out R^k caused by traffic jams or red traffic lights.

4.3 Pickup Location Clustering

Given an individual taxi's trajectory $\Re$ having multiple slow pickup events, the output of the PEA algorithm, i.e., the sub-trajectory set ω, contains a number of sub-trajectories, i.e., $\{R^k | k = 1, 2, \cdots\}$. For each sub-trajectory R^k, i.e., $p_{s_k} \rightarrow \cdots \rightarrow p_{e_k}$, we compute a central GPS location $(\bar{c}_{lat}, \bar{c}_{lon})$ by averaging their latitude coordinates and the longitude coordinates. Accordingly, we have a GPS location set $c = \{(\bar{c}^k_{lat}, \bar{c}^k_{lon}) | k = 1, 2, \cdots\}$ derived from the sub-trajectory set ω.

After running the PEA algorithm on all taxis' trajectories respectively, we have a sub-trajectory set $W = \{\omega^j | j = 1, 2, \cdots\}$, where ω^j is the j^{th} taxi's sub-trajectory set ω. Accordingly, we have the GPS location set $C = \{c^j | j = 1, 2, \cdots\}$, where c^j is the j^{th} taxi's GPS location set c.

Given all taxis' GPS location set C, we run it with the density-based clustering method DBSCAN [5], which is an effective way to discover high density clusters and remove noises. We then compute the centroid of all the found clusters, and each centroid is the detected taxi queue spot. Given the fact that people always queue for taxis at the places where taxis usually take passengers, it is reasonable to assume the detected taxi queue spots are also the possible spots where taxi passengers queue up. We thus call an obtained centroid *queue spot* rather than taxi or passenger queue spot.

The GPS location set C is normally a large dataset. When running the DBSCAN algorithm on it, we need to carefully select its parameters and strive to reduce the runtime complexity (e.g., using the R-Tree based or grid based spatial index). We will address all of these implementation issues in section 6. On the other hand, many other advanced density-based clustering methods can also be considered and introduced [13].

5. QUEUE CONTEXT DISAMBIGUATION

5.1 Wait Time Extraction

Given $W(r)$ is the extracted pickup event sub-trajectory set W for queue spot r, it consists of a large number of sub-trajectories from different taxis. For each sub-trajectory in $W(r)$, the corresponding wait time is the time interval between the wait start time and the wait end time. We therefore present a simple algorithm, called wait time extraction (WTE) algorithm: its input is $W(r)$ for a queue spot r and output is the taxi wait time set $Y(r) = \{t^m_{end} - t^m_{start} | m = 1, 2, \cdots\}$, where t^m_{end} and t^m_{start} are the wait start time and wait end time of the m^{th} taxi. The complete algorithm is shown in Algorithm 2.

For each sub-trajectory $R(s, e)$ in $W(r)$, the WTE set the wait start time t_{start} to the timestamp when the first FREE, ONCALL or ARRIVED state appears. However, if any PAYMENT is detected thereafter, it resets t_{start} to the timestamp when the subsequent FREE state appears. After the wait start time t_{start} is determined, the wait end time t_{end} is set to the timestamp when the first POB state appears. Finally, the time interval between the wait start time and the end time is added into the taxi wait time set $Y(r)$, i.e., $\{t^m_{wait} | m = 1, 2, \cdots\}$.

5.2 Time Slot with Pickup Event Feature

We divide the time domain into L continuous time slots, where time slot T^j $(1 \leq j \leq L)$ starts at t^{j-1} and ends at

Algorithm 2 Wait Time Extraction Algorithm

```
Input: $W(r)$ for queue spot $r$.
Output: Taxi Wait time set $Y(r)$.
 1: $t_{start} \leftarrow null$; $t_{end} \leftarrow null$;
 2: for each sub-trajectory $R(s,e)$ in $W(r)$ do
 3:   for $i = s_m \rightarrow e_m$ do
 4:     if $p_{i.state} = \{FREE\ or\ ONCALL\ or\ ARRIVED\}$
        and $t_{start} = null$ then
 5:       $t_{start} \leftarrow p_{i.ts}$;
 6:     else if $p_{i.state} = PAYMENT$ and $t_{start} \neq null$
        then
 7:       $t_{start} \leftarrow null$; $t_{end} \leftarrow null$;
 8:     else if $p_{i.state} = POB$ and $t_{start} \neq null$ and $t_{end} =$
        $null$ then
 9:       $t_{end} \leftarrow p_{i.ts}$;
10:     end if
11:   end for
12:   if $t_{start} \neq null$ and $t_{end} \neq null$ then
13:     $Y(r).Add(t_{end} - t_{start})$
14:   end if
15:   $t_{start} \leftarrow null$; $t_{end} \leftarrow null$;
16: end for
```

t^j. The time t^0 and t^j are the start time and end time of the time domain. Accordingly, set $Y(r)$ can be divided into L partitions, where $Y(r)^j = \{t^m_{wait} | t^{j-1} \leq t^m_{start} < t^j\}$ and $1 \leq j \leq L$.

Given $Y(r)^j$ consists of multiple t^m_{wait}, we have their arithmetic mean, denoted by $\bar{t}_{wait}(r)^j$, as the average taxi wait time over time slot T^j. When computing $\bar{t}_{wait}(r)^j$, we only consider all street jobs' wait time, i.e., t^m_{start} set by the timestamp of FREE, as a booking job's wait time mainly depends on a specific booking passenger's individual arrival time.

Given the arrival number of FREE taxi over time slot T^j, denoted by $N_{arr}(r)^j$, and time slot length $t^j - t^{j-1}$, we have the average arrival rate of FREE taxis over time slot T^j: $\bar{\lambda}(r)^j = \frac{N_{arr}(r)^j}{t^{j+1} - t^j}$. According to Little's Law [7], which relates average arrival rate, average wait time and average queue length, we have the average FREE taxi queue length over time slot T^j: $\bar{L}(r)^j = \bar{t}_{wait}(r)^j * \bar{\lambda}(r)^j$.

Meanwhile, the taxi departure interval can be computed by $t^{m+1}_{end} - t^m_{end}$, where t^m_{end} and t^{m+1}_{end} are the consecutive wait end time in $Y(r)^j$. Accordingly, we have the arithmetic mean of all the departure intervals, denoted by $\bar{t}_{dep}(r)^j$ over time slot T^j. When computing $\bar{t}_{dep}(r)^j$, we consider the departure time intervals of all the departed taxis, i.e., both street job ones and booking job ones. Thus $\bar{t}_{dep}(r)^j$ depicts the departure rate of all departed taxis over time slot T^j.

Finally, we have a 5-tuple to depict time slot T^j of spot r: $\varphi(r)^j = \left\langle \bar{t}_{wait}(r)^j, N_{arr}(r)^j, \bar{L}(r)^j, \bar{t}_{dep}(r)^j, N_{dep}(r)^j \right\rangle$, where $\bar{t}_{wait}(r)^j$ and $N_{arr}(r)^j$ depict the FREE taxi arrival activities, $\bar{L}(r)^j$ gives the FREE taxi queue length, $\bar{t}_{dep}(r)^j$ and $N_{dep}(r)^j$ depict all taxis' departure activities. All the 5 variables are pickup event features.

As described earlier, the MDT logging is an event-driven action, where it records the exact moment that the taxi state switch to FREE, ARRIVED, PAYMENT or POB. Therefore, the values in the 5-tuple are highly accurate and valid.

5.3 Queue Context Disambiguation

Given the 5-tuple for each time slot at a queue spot, we propose a conceptually simple and easily implemented algorithm, called queue context disambiguation (QCD) algorithm, to identify the different queue types defined in Table 3. The basic idea behind the QCD algorithm is that: a passenger queue may exist when taxi pickup events frequently and continuously occur; besides, a passenger queue may also exist when a considerably high proportion of booked taxis appear, even though the frequency of taxi pickup events is not high.

The main input of QCD is the derived 5-tuple feature set for all time slots $\Omega(r) = \{\varphi(r)^j | j = 1, 2, \cdots, L\}$, and the output is the labeled time slots, i.e., C_1, C_2, C_3 and C_4. The complete algorithm is shown in Algorithm 3.

Algorithm 3 Queue Context Disambiguation Algorithm

```
Input: $\Omega(r)$, thresholds $\eta_{wait}$,$\eta_{dep}$,$\tau_{arr}$,$\tau_{dep}$,$\eta_{dur}$,$\tau_{ratio}$.
Output: Labeled $T^j$, where $1 \leq j \leq L$.
 1: Routine 1:
 2: for $j = 1 \rightarrow L$ do
 3:   if $\bar{L}(r)^j < 1$ then
 4:     if $N_{arr}(r)^j \geq \tau_{arr}$ and $\bar{t}_{wait}(r)^j < \eta_{wait}$ then
 5:       Label $T^j$ to $C_2$;
 6:     else if $N_{arr}(r)^j < \tau_{arr}$ and $\bar{t}_{wait}(r)^j \geq \eta_{wait}$ then
 7:       Label $T^j$ to $C_4$;
 8:     end if
 9:   end if
10:   if $\bar{L}(r)^j \geq 1$ then
11:     if $N_{dep}(r)^j \geq \tau_{dep}$ and $\bar{t}_{dep}(r)^j < \eta_{dep}$ then
12:       Label $T^j$ to $C_1$;
13:     else if $N_{dep}(r)^j < \tau_{dep}$ and $\bar{t}_{dep}(r)^j \geq \eta_{dep}$ then
14:       Label $T^j$ to $C_3$;
15:     end if
16:   end if
17: end for
18: Routine 2:
19: for $j = 1 \rightarrow L$ do
20:   if $T^j$ not labeled and $N_{dep}(r)^j * \bar{t}_{dep}(r)^j > \eta_{dur}$ and
      $\frac{N_{arr}(r)^j}{N_{dep}(r)^j} < \tau_{ratio}$ and $\bar{L}(r)^j \geq 1$ then
21:     Label $T^j$ to $C_1$;
22:   else if $T^j$ not labeled and $N_{dep}(r)^j * \bar{t}_{dep}(r)^j > \eta_{dur}$
      and $\frac{N_{arr}(r)^j}{N_{dep}(r)^j} < \tau_{ratio}$ and $\bar{L}(r)^j < 1$ then
23:     Label $T^j$ to $C_2$;
24:   end if
25: end for
```

In general, the proposed QCD algorithm consists of 2 routines to identify the queue type, which use different criteria to label the time slots. In Routine 1, it firstly examines whether a taxi queue exists during the given time slot by checking the queue length value $\bar{L}(r)^j$. When a taxi queue does not exist, i.e., $\bar{L}(r)^j < 1$, it further analyzes the average taxi wait time $\bar{t}_{wait}(r)^j$ and arrival taxi number $N_{arr}(r)^j$: a considerably large taxi arrival number with a small average taxi wait time value indicates a passenger queue exist, and thus it labels the time slot as C_2; On the other hand, a small taxi arrival number with a considerably large average taxi wait time value indicates no passenger queue exist, and thus it labels the time slot as C_4. In other words, the taxi arrival number together with the average taxi wait time value serves as the indicator of a passenger queue. When a taxi queue exists, i.e., $\bar{L}(r)^j >= 1$, it analyzes the average taxi departure interval $N_{dep}(r)^j$ and departure taxi number$N_{dep}(r)^j$: a large taxi departure number with a notably small average taxi departure interval value indicates a passenger queue exist, and thus it labels the time slot as C_1; On the other hand, a small taxi departure number with a notably large average taxi leave interval value indicates no passenger queue exist, and thus it labels the time slot as C_3. Note that the average taxi wait time value is no longer a good indicator of a passenger queue when a taxi queue exist, i.e., $\bar{L}(r)^j >= 1$, as the queuing delay becomes significant and even a dominant factor; on the other hand, the average taxi departure interval value is not a good indicator of a passenger queue when a taxi queue does not exist, i.e., $\bar{L}(r)^j < 1$, as some taxis may depart with booking passengers, and the corresponding leave intervals are only determined by the booking passengers' arrival time.

In Routine 2, the QCD algorithm continues to label the time slots that can not be identified in Routine 1: it mainly uses the ratio $\frac{N_{arr}(r)^j}{N_{dep}(r)^j}$, i.e., the ratio of the arrival FREE taxi number to the total departure taxi number (street jobs + booking jobs) over the given time slot, to infer whether a passenger queue exist. More specifically, a small value of $\frac{N_{arr}(r)^j}{N_{dep}(r)^j}$ essentially means a large portion of ONCALL taxis depart from this queue spot, which strongly indicates the difficulty to get a FREE taxi at the current spot and time slot. Meanwhile, if the departure events occur over a long time period, i.e., $N_{dep}(r)^j * \bar{t}_{dep}(r)^j > \eta_{dur}$, the QCD algorithm would infer the existence of a passenger queue, and accordingly labels the given time slot to C_1 or C_2 according to the taxi queue length value $\bar{L}(r)^j$. Note that in Singapore, people usually prefer hailing down a *FREE* taxi rather than booking a taxi, as they have to pay a compulsory 3 to 4 Singapore dollars' taxi booking fee, which easily takes up more than 30% of a single-trip taxi fare.

The threshold values used in the proposed QCD algorithm need to be properly set, and different queue spots may have different threshold values: for example, a queue spot located at a hospital might have distinct threshold values from the one in the airport. We will illustrate how to determine these values in section 6.

6. EMPIRICAL EVALUATION

6.1 Queue Spot Detection Experiment

6.1.1 Data Preprocessing

Our entire dataset contains about 15000 taxis' MDT logs, which occupy around 60% of the total taxis in Singapore. In general, the 15000 taxis generate around 12.38 million daily MDT log records, and each MDT generate 848 daily MDT log records. It is natural that such a large MDT log dataset contains some errors, and the main error types are: (1) improper/missing taxi states; (2) record duplication; (3) GPS coordinates outside Singapore or in inaccessible zones.

Firstly, the standard taxi state transition diagram is given in Fig. 3, but the taxi states in some MDT logs appear at improper places. For example, a FREE state is found between the two PAYMENT states in many MDT logs. We found that it is a software bug caused by the clock synchronization between the old version MDT device and the taximeter. Another common issue is some intermediate taxi states, e.g., ARRIVED, NOSHOW or STC, are missing and lost. Two possible reasons are identified: 1) these intermediate states sometimes last only several seconds and then switch to another state before the MDT logging thread tracks them down. 2) logging some intermediate states requires taxi driver manually press some specific buttons on the MDT touch screen, but some drivers omit this step either purposely or accidentally. Secondly, the duplicate records in the MDT logs are mainly caused by the re-transmission of the GPRS messages between MDT and the backend service. Thirdly, the GPS coordinates errors are normally caused by the urban canyon effect [3].

In short, we remove the above described erroneous records from the raw MDT log dataset, which occupy around 2.8% of the total MDT log records.

6.1.2 Experiment Setup and Parameter Selection

We use the daily MDT logs from all 15000 taxis for the queue spot detection. Firstly, we run the proposed PEA algorithm on the 15000 taxis' daily individual trajectories, where 10 km per hour is as the speed threshold η_{sp}, and successfully extract more than 264000 pickup event sub-trajectories for each single day. Each extracted sub-trajectory provides us one central GPS location, and accordingly we have around 264000 GPS locations, i.e., the GPS location set C, for the DBSCAN clustering. Running the DBSCAN clustering algorithm on such a large-size point set is significantly slow due to its $O(n^2)$ complexity. Therefore, we simply divide Singapore into 4 rectangular zones based on their different characteristics, i.e., Central, North, West and East, as illustrated in Fig. 5. The central zone covers Singapore's central business district(CBD) and most of tourist attractions. The other 3 zones are typically residential and industrial areas with a few tourist attractions. Therefore, the GPS location set C is further divided into 4 subsets, and we run the DBSCAN clustering algorithm on each subset respectively.

Moreover, properly choosing the two parameters of DBSCAN, i.e., eps ε_d and min-points p_d, is not a trivial issue: ε_d specifies the maximum radius of the neighborhood and p_d sets the minimum number of points in an eps-neighborhood of the point. An unduly small ε_d or an overly large p_d may lead a large part of the data points cannot be clustered, while an overly large ε_d or an unduly small p_d would merge different clusters into one. Fig. 6 shows the number of the detected queue spot with respect to different ε_d and p_d. We see that small ε_d values (e.g., 10 meters) or large p_d values (e.g., 100 points) result that only a few number of queue spots are detected and many actual ones are neglected. On the other hand, large ε_d values (e.g., 20 meters) or small p_d values (e.g., 25 points) would easily merge adjacent queue spots and meanwhile bring many insignificant queue spots.

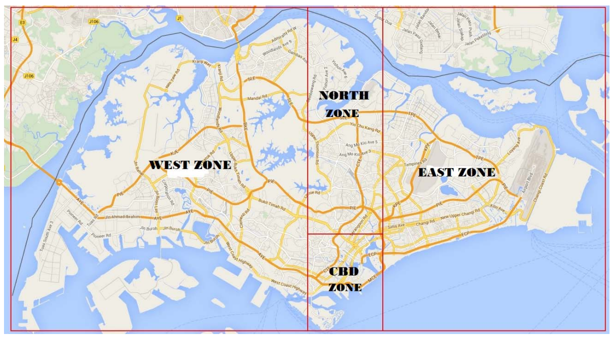


Figure 5: Four Rectangular Zones

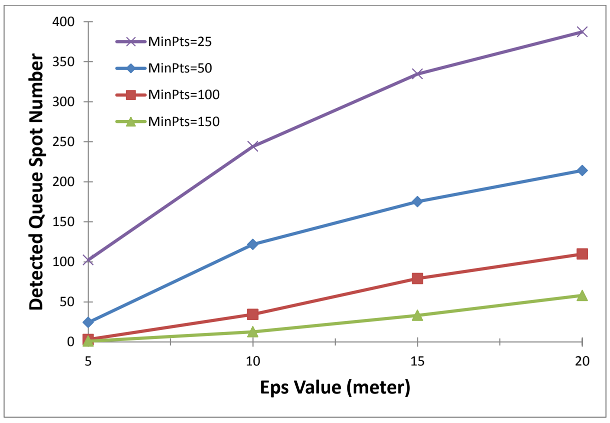


Figure 6: DBSCAN Performance with Different Parameter Pairs

By carefully comparing the DBSCAN clustering results, we finally set its parameter ε_d and p_d to 15 meters and 50 points respectively when processing the daily Singapore taxis' MDT log dataset. Roughly speaking, the selected two parameters allow detect the queue spot that has more than 50 taxi pickup events within its proximity of 15 meters. Note that the selected parameters are used to process the MDT log dataset on the daily basis, and for different time durations, e.g., one week's MDT log dataset, we may have to change and reset the DBSCAN parameters.

6.1.3 Queue Spot Detection Results and Analysis

With the selected DBSCAN parameters, totally around 180 queue spots are detected in the four zones of Singapore, as illustrated in Fig. 7. We analyze and manually label the detected queue spots by locating their GPS coordinates on Google Maps with its Street View feature. It shows that most of the detected queue spots are located nearby the public facilities or landmarks. Table 4 summarizes the result: almost half of the queue spots are nearby Singapore Mass Rapid Transit (MRT) stations or bus stations, around 20% of them are located nearby the shopping malls, hotels or office buildings. Only around 5% of the detected queue spots do not have any significant nearby facility or landmark.

We compare the queue spot detection results with the taxi stand locations, which are sets up by Singapore Land Authority (LTA): there are 31 taxi stands with more than 3 taxi

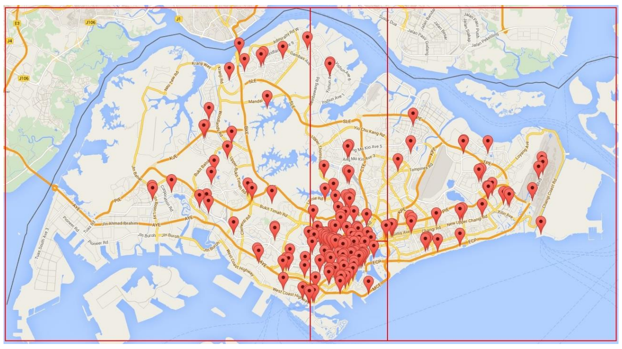


Figure 7: Detected Queue Spots in Singapore

Table 4: Landmark Nearby the Detected Queue Spots

Nearby Facility or Landmark	Detected Spots Percentage
MRT & BUS station	48.3%
Shopping Mall & Hotel	11.8%
Office Building	9.6%
Hospital & School	8.4%
Tourist Attraction	6.2%
Airport & Ferry Terminal	5.6%
Industrial and Residential Area	4.5%
Unidentified	5.6%

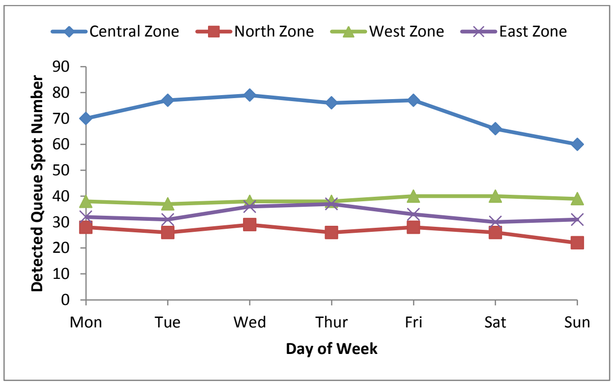


Figure 8: Queue Spot Number in Different Zones and Days

parking lots in the CBD area, and 30 of them are correctly detected with the average location error only 7.6 meters (possibly caused by the GPS error). More importantly, more than 15 queue spots in this area, which are not labeled by LTA, are busy enough and even have more daily pickups than many taxi stands.

We further compare the detected queue spot number in different zones and days of week. Fig. 8 shows that central zone has the largest number of queue spots, although it only occupies around 6% of the total area. The main reason is that most of the high-rise office buildings, shopping malls and tourist attractions in Singapore are located in this zone. More importantly, Fig. 8 shows that queue spot numbers in all zones do not have a high fluctuation on different

Table 5: Hausdorff Distance between Queue Spot Sets on Different Days of Week (Meter)

Hausdorff distance	Mon	Tue	Wed	Thurs	Fri	Sat	Sun
Mon	0	57.076	42.958	59.475	45.531	104.61	143.27
Tue	57.076	0	44.768	34.649	54.293	117.41	141.07
Wed	42.958	44.768	0	41.429	54.311	106.71	139.87
Thurs	59.475	34.649	41.429	0	57.721	111.58	133.21
Fri	45.531	54.293	54.311	57.721	0	81.125	119.41
Sat	104.61	117.41	106.71	111.58	81.125	0	67.111
Sun	143.27	141.07	139.87	133.21	119.41	67.111	0

week days (from Monday to Friday), while the queue spot number slightly drops down on Saturday and Sunday in the central zone. It is probably caused by fewer local working people traveling in the CBD area during the weekend. The queue spot number during weekend does not drop significantly, as still many people go to the shopping malls and tourist attractions in the central zone.

To gain further insight into the queue spot detection results on different days of week, we adopt the modified Hausdorff distance [4] to evaluate their similarity and stability. Hausdorff distance (or called Pompeiu-Hausdorff distance) has been widely used to measure the similarity of two point sets in object matching. Briefly speaking, it is the maximum distance of a point set to the nearest point in the other point set. Therefore, the Hausdorff distance between the detected queue spot sets illustrates whether the queue spot sets on different days of week are stable and how far they are from each other. Table 5 shows the Hausdorff distance between the detected queue spot sets in meter, where each set consists of all the four zones' detected queue spots.

Based on the definition of the Hausdorff distance, the value "0" means the two spot sets are overlapped exactly, and a large distance value means big mismatch between the two queue spot sets. From Table 5, we see that the Hausdorff distances between any two queue spot sets are only around 50 meters on the week days and around 67 meters on the weekend days: it indicates that the detected queue spots match quite well and thus the queue locations are normally stable. When we compare a week day queue spot set with a weekend queue spot set, e.g., Monday and Sunday, the corresponding Hausdorff distance easily increases to around 130 meters, but it is still a relatively low value given Singapore an area with 50 kilometers long and 26 kilometers wide.

In short, the proposed queue spot detection method with the properly selected parameters effectively detects the Singapore island-wide queue spots with a high accuracy and stability.

6.2 Context Disambiguation Experiment

6.2.1 Feature Preparation and Threshold Selection

For each detected queue spot, say r, we have its corresponding pickup event sub-trajectory set $W(r)$, which usually consists of a number of sub-trajectories ranging from 100 to 500 daily. Table 6 illustrates the daily average number at a queue spot in different zones and days. We see that a queue spot has around 200 sub-trajectories on average on a week day, and the number slightly goes up on a weekend day. Meanwhile, the average number in the east zone is always higher than the other 3 zones: it is probably caused by a large number of pickup events at Singapore's Changi international airport, which is located at the east zone.

Table 6: Average Pickup Event Number

Avg. Sub-trajectory Number	Central	North	West	East
Working Day	217.5	165.5	223.3	267.2
Weekend Day	251.6	172.3	198.1	305.8

We conduct the experiment on a daily basis and divide one day into 48 fixed-size time slot. Each time slot thus takes 1800 seconds, e.g., 00:00 to 00:30 or 18:30 to 19:00. For each detected queue spot, say r, we run the WTE algorithm on its pickup event sub-trajectory set $W(r)$, and accordingly derive the 5-tuple feature set for all 48 time slots, i.e., $\Omega(r) = \{\varphi(r)^j | j = 1, 2, \cdots, 48\}$.

Before running the QCD algorithm on $\Omega(r)$, we need to determine the 6 threshold values used in the algorithm. For each queue spot, we select its top 20% shortest wait time values and top 20% shortest departure intervals, which can commonly depict taxi wait and departure events when the passenger queue exists. We thus use their average values as the threshold η_{wait} and η_{dep} respectively. Accordingly, we set the threshold τ_{arr} and τ_{dep} to $\frac{1800}{\eta_{wait}}$ and $\frac{1800}{\eta_{dep}}$ respectively, where 1800 is the predetermined time slot length in seconds. Meanwhile, the threshold η_{dur} is set to 90% of the current time slot length, namely 1620 seconds. To determine the threshold τ_{ratio}, we calculate the daily ratio of the total street job number to the total job number (street jobs + booking jobs) in different zones and days of week, and then set the threshold τ_{ratio} to the corresponding ratio value, e.g., 0.84 is the average ratio value in the central zone on Sunday. The taxi state transition knowledge, as illustrated in Fig. 3, is directly used to derive and separate booking jobs and street jobs from the MDT logs.

Lastly, given the fact that our dataset, i.e., around 15000 taxis' MDT logs, only occupies 60% of the total operating

Table 7: Proportion of Different Queue Types

Queue Type	Percentage in All Time Slots
C_1	30.1%
C_2	11.7%
C_3	8.6%
C_4	33.1%
Unidentified	16.5%

taxis in Singapore, we increase the feature values $N_{arr}(r)^j$, $\bar{L}(r)^j$ and $N_{dep}(r)^j$ by multiplying an amplification factor 1.667 and decrease the feature value $\bar{t}_{dep}(r)^j$ by multiplying 0.6 for all time slots.

6.2.2 *Experiment Results and Analysis*

Based on the extracted 5-tuple feature set and the determined threshold values, we run the QCD algorithm on 25 randomly selected queue spots respectively. Table 7 summarizes the queue type identification results: nearly 84% time slots in total are successfully labeled as C_1 to C_4 respectively, among which 30% time slots are labeled as C_1, namely taxi queue and passenger queue concurrently exist, while 33% time slots are labeled as C_4, namely no any taxi queue or passenger queue during these time slots. Meanwhile, only 11.7% time slots are labeled as C_2, namely only passenger queue exists, and 8.6% time slots are labeled as C_3, namely only taxi queue exists. It shows that such two situations do not occur as frequently as C_1 or C_4. Besides, around 16% time slots cannot be identified by the QCD algorithm due to their insignificant features. For example, during a time slot T^j that no taxi queue exist, i.e., queue length $\bar{L}(r)^j < 1$, only several taxis, e.g., 7 or 8, arrive and depart with a moderate average wait time value; meanwhile, there is no significant number of *ONCALL* taxis arrive and leave. In such cases, the QCD algorithm does not label the time slot to any predefined type, but simply label it as unidentified or insignificant type.

We further compare the queue type identification results on different days of week. Fig. 9 shows that during the five week days, all queue types' proportion do not have a high fluctuation. However, during the two weekend days, especially on Sunday, the proportion of C_4 significantly goes up from 30% to around 40%, and meanwhile the proportions of C_2 and the unidentified time slots drop down. One possible explanation is that during weekend days fewer business and working people traveling leads to more C_4 time slots and accordingly fewer C_2 and unidentified time slots. Fig. 9 also shows that the proportion of C_3 slightly goes down and C_1 generally maintains its proportion over a whole week.

To further validate the queue type identification results, we collect the waiting taxi number from an independent vehicle monitor system [14], which is set up for continuously observing the vehicle number inside a taxi stand area (normally a predefined polygon). The monitor system updates the vehicle number every 60 seconds via its RESTful web service, which can be used as a good indicator for the existence of a taxi queue. On the other hand, we take the failed taxi booking records from the taxi operator's backend database.

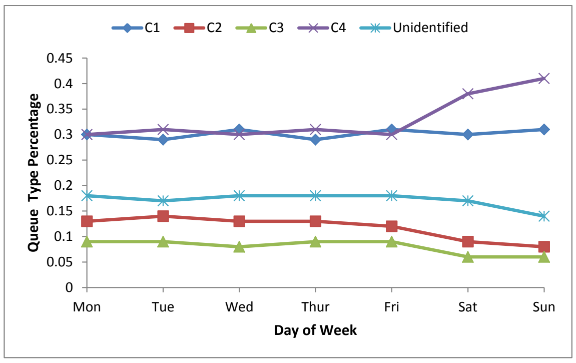


Figure 9: Proportion of Queue Type in Different Days of Week

Table 8: Average Number of Taxis and Failed Bookings

Queue Type	Avg. Taxi Number	Avg. Failed Booking Number
C_1	6.13	0.35
C_2	1.35	4.29
C_3	3.26	0.13
C_4	0.32	0.73
Unidentified	1.56	0.24

A failed taxi booking means the booking request has been successfully dispatched to all the nearby taxis, but the passenger finally fails to get a taxi due to no taxi available inside the dispatching circle centered at the pickup location with radius 1 kilometer. Frequently failed bookings over a short period at the same pickup location indicate that the passenger's current demand is much higher than the taxi's current supply, and thus can be used to imply the existence of a passenger queue.

Table 8 shows the average taxi number obtained from the vehicle monitor system and the average failed bookings number obtained from the taxi operator during the labeled time slots. We see that the average taxi numbers of C_1 and C_3 are notably higher than the corresponding values of C_2 and C_4: it suggests a high chance of a taxi queue occurring during the time slots labeled as C_1 and C_3. Meanwhile, the average failed booking numbers of C_2 is significantly higher than the others: it strongly indicates a high chance of a passenger queue occurring during the time slots labeled as C_2. Note that the time slots labeled as C_1 do not have many failed bookings: it is probably because many available taxis queuing at or nearby the queue spot. In short, the failed booking data combined with the information from an independent vehicle monitor system, at least to some extend, validates the queue type identification results.

6.2.3 *A Sample Case: Lucky Plaza Queue Spot*

To demonstrate the queue type identification results from the individual perspective, we take one queue spot, which is detected nearby the main entrance of Singapore Lucky Plaza, as an illustrative example. Lucky Plaza is a shopping center located at Orchard Road, which is a famous retail and entertainment hub in Singapore. We simply pick up a Sunday's queue type identification results at the Lucky

Table 9: A Sample Queue Type Identification Result

Queue Type	C1	C2	C3	C4	Unidentified
Time Slot	00:00 --- 00:30 09:30 --- 10:00 11:00 --- 15:30 17:30 ---19:30	15:30 --- 17:30 19:30 --- 20:00	00:30 --- 1:30 10:00 --- 11:00 20:00 --- 21:30	1:30 --- 08:30 21:30 --- 23:30	8:30 --- 9:30 23:30 --- 24:00

Plaza queue spot, and summarize them in Table 9.

From Table 9, we see that during the early midnight the queue type C_1 (from 00:00 to 00:30) and C_3 (from 00:30 to 01:30) are identified, which means the concurrent passenger queue and taxi queue occur first and then only the taxi queue left. After that, the queue type C_4 lasts 7 hours (from 01:30 to 8:30), meaning no taxi queue or passenger queue until the early morning. During the peak shopping hours (from 11:00 to 20:00), the queue type shifts between C_1 and C_2, meaning either the concurrent passenger queue and taxi queue or only the passenger queue at Lucky Plaza. After the peak shopping hours, the queue type switches back to C_4 (from 21:30 to 23:30), i.e., no taxi queue or passenger queue. We conducted a short term study at the Lucky Plaza queue spot: the actual queue variance pattern fits well with the above described queue type identification result. For example, during the early midnight, most of people, who just leave the nearby night clubs, usually wait taxis at this queue spot, which explains why the queue type changes to C_1 or C_3 during such time slots.

7. DISCUSSION

7.1 A Real World Deployment

To better serve the stakeholders of our solution (the government agencies, public commuters and taxi operator), we implemented a practical taxi queue detection and analysis system, which consists of a backend queue analytic engine and a web-based frontend user interface.

Within the queue analytic engine, the queue spot detection module uses the relatively long-term historical dataset to extract queue spots. Based on the evaluation results, the queue spot sets in Singapore show some differences between week days and weekend days. Thus, we simply use historical week days' dataset to extract queue spots for a week day, and use historical weekend days' dataset to extract queue spots for a weekend day. In the current implementation, the queue spot detection module collects the most recent 5 week days' dataset and 2 weekend days' dataset to extract and update the corresponding queue locations. The queue context disambiguation module currently mainly runs on the short-term historical dataset to generate the queue type transition reports for week day and weekend day respectively. The historical dataset size and the update frequency for both modules can be configured by system users.

The web-based user interface provide users a simple way to understand and access the queue detection and analysis results. When a user opens the query page, it firstly shows all the latest detected queue spots in Singapore on the Google Map. By zooming into a region and placing the mouse on a specific queue spot, the user can see the identified queue type at the selected time slot together with the nearby facility name as shown in Fig. 10. If necessary, the user can

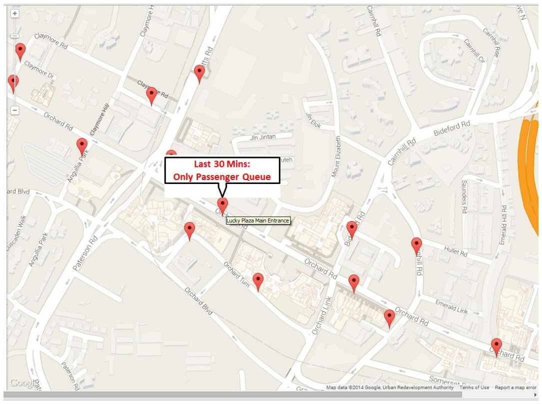


Figure 10: User Interface of the Deployed System

further query the long-term queue type transition reports and save it into the database or a text file.

The backend queue analytic engine is mainly implemented in Java and uses Java database connectivity (JDBC) to retrieve the readily available MDT logs in a PostgreSQL database system. The frontend user interface is developed based on Google Map Javascript API.

7.2 Interesting Findings

Driver Behavior: the taxi *BUSY* state, as describe in Table 1, is designed for taxi drivers when they temporarily unavailable due to personal reasons. However, we find that during the time slots of C_1 and C_2, especially C_2 (namely only passenger queue), a number of taxis enter the queue spots with a *BUSY* state and then quickly leave with a *POB* state. Such a phenomenon indicates that some taxi drivers only pick up their favorite passengers and deny the others by using the *BUSY* state as an excuse. We are currently further investigating on this issue and the possible solutions.

Sporadic Queue Spot: although the detected queue spots in Singapore show an overall high stability, there are a few exceptions. For example, a queue spot inside the west zone periodically appears only on every Sunday (occasionally on Saturday) but never shows during week days. By manually labeling this queue spot, we locate it at a local leisure park, which is not a quite famous tourist attraction in Singapore but only a popular place for the local family during the weekend. Another example is that an individual queue spot is detected only on a specific Sunday at Sentosa island, and through the local newspaper we find that it is caused by a company's anniversary celebration event. These examples not only show that the sporadic queue spots can be precisely captured by our solution, but also demonstrate that meaningful semantics behind both regular and sporadic queue spots can be further explored.

8. RELATED WORK

8.1 Queue Sensing and Detection

Queuing theory [6] has been extensively studied and widely applied (e.g., in the wireless communications), while there is limited research work on detection and analysis of the real world queuing behaviors. The early studies [10] conduct video analytics to detect human queuing activity based on stationary cameras. Recently, people start using smartphones [12] and sensor networks [11] to sense and detect human and vehicle queues. All the existing solutions require fixed infrastructures with maintenance overheads and additional costs.

8.2 Taxi Trace Analytics

Driven by the availability of abundant information in taxi traces, especially the GPS location information, mining taxi traces has received massive attentions from both academia and industry in recent years. The existing work can be generally classified into three categories: 1) mining taxi traces to study the city population movement patterns and behaviors [9, 15]; 2) using taxi traces as a probe to infer or predict traffic conditions for city road networks [1, 8]; 3) mining taxi traces to discover and sense human or vehicle's special events and behaviors [16, 17]. For example, the authors in [16] use taxi traces to sense refueling behavior and citywide petrol consumption. Our work can be classified into the third category, and we refer the interested readers to a good survey [2] for taxi traces analytics.

To the best of our knowledge, there is no previous work using taxi traces to conduct the real world queue analytics for both taxis and their passengers.

9. CONCLUSION AND FUTURE WORK

By leveraging on the event-driven MDT logs and the taxi state transitions, we design and implement a practical system to effectively detect the queue spots and identify the queue context in Singapore. The extensive evaluation results show the feasibility and the accuracy of the deployed queue analytic engine.

The deployed system and the queue context analysis results lay a solid foundation for future work:

- Integrate the queue analytic information into the existing MDT system to conduct recommendations for taxi drivers, e.g., suggesting recent emerging passenger queue spots.
- Periodically publish both taxi queue and passenger queue information to the public and help to reduce queuing events in the city.
- Work with LTA to set up new taxi stands at the busy queuing spots and improving the existing facilities.

Lastly, we would like to highlight that the nature of this work is applicable to not only Singapore but also other densely populated cities, given their taxis equipped with the MDT-like telematics devices.

10. REFERENCES

[1] J. Aslam, S. Lim, X. Pan, and D. Rus. City-scale traffic estimation from a roving sensor network. In *Proc. ACM Conference on Embedded Network Sensor Systems (SenSys)*, 2012.

[2] P. S. Castro, D. Zhang, C. Chen, S. Li, and G. Pan. From taxi gps traces to social and community dynamics: A survey. *ACM Comput. Surv.*, 46(2):17:1–17:34, Dec. 2013.

[3] M. Dottling, F. Kuchen, and W. Wiesbeck. Deterministic modeling of the street canyon effect in urban micro and pico cells. In *Proc. IEEE International Conference on Communications (ICC)*, pages 36–40, June 1997.

[4] M.-P. Dubuisson and A. Jain. A Modified Hausdorff Distance for Object Matching. In *Proc. IAPR International Conference on Computer Vision and Image Processing*, pages 566–568, Oct. 1994.

[5] M. Ester, H. peter Kriegel, J. S, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proc. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 1996.

[6] D. Gross, J. F. Shortle, J. M. Thompson, and C. M. Harris. *Fundamentals of Queueing Theory*. Wiley-Interscience, New York, NY, USA, 2008.

[7] J. D. C. Little. A Proof for the Queuing Formula: L = λW. *Operations Research*, 9(3):383–387, 1961.

[8] S. Liu, Y. Liu, L. M. Ni, J. Fan, and M. Li. Towards mobility-based clustering. In *Proc. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2010.

[9] L. Moreira-Matias, J. Gama, M. Ferreira, J. Mendes-Moreira, and L. Damas. Predicting taxi-passenger demand using streaming data. *IEEE Transactions on Intelligent Transportation Systems*, 14(3):1393–1402, Sept 2013.

[10] J. Segen. A camera-based system for tracking people in real time. In *Proc. International Conference on Pattern Recognition (ICPR)*, 1996.

[11] R. Sen, A. Maurya, B. Raman, R. Mehta, R. Kalyanaraman, N. Vankadhara, S. Roy, and P. Sharma. Kyun queue: A sensor network system to monitor road traffic queues. In *Proc. ACM Conference on Embedded Network Sensor Systems(SenSys)*, 2012.

[12] Y. Wang, J. Yang, Y. Chen, H. Liu, M. Gruteser, and R. P. Martin. Tracking human queues using single-point signal monitoring. In *Proc. ACM Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2014.

[13] I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. San Francisco: Morgan Kaufmann, 2005.

[14] W. Wu, W. S. Ng, S. Krishnaswamy, and A. Sinha. To taxi or not to taxi? - enabling personalised and real-time transportation decisions for mobile users. In *Proc. IEEE International Conference on Mobile Data Management (MDM)*, pages 320–323, July 2012.

[15] N. J. Yuan, Y. Zheng, L. Zhang, and X. Xie. T-finder: A recommender system for finding passengers and vacant taxis. *IEEE Transactions on Knowledge and Data Engineering*, 25(10):2390–2403, 2013.

[16] F. Zhang, D. Wilkie, Y. Zheng, and X. Xie. Sensing the pulse of urban refueling behavior. In *Proc. ACM International Joint Conference on Pervasive and Ubiquitous Computing(UbiComp)*, 2013.

[17] Y. Zheng, L. Zhang, X. Xie, and W.-Y. Ma. Mining interesting locations and travel sequences from gps trajectories. In *Proc. ACM Conference on World Wide Web (WWW)*, New York, NY, USA, 2009.

Data Ingestion in AsterixDB

Raman Grover, Michael J. Carey

Department of Computer Science,
University of California- Irvine, CA, 92697
{ ramang, mjcarey}@ics.uci.edu

ABSTRACT

In this paper we describe the support for data ingestion in AsterixDB, an open-source Big Data Management System (BDMS) that provides a platform for storage and analysis of large volumes of semi-structured data. Data feeds are a new mechanism for having continuous data arrive into a BDMS from external sources and incrementally populate a persisted dataset and associated indexes. We add a new BDMS architectural component, called a *data feed*, that makes a Big Data system the caretaker for functionality that used to live outside, and we show how it improves users' lives and system performance.

We show how to build the *data feed* component, architecturally, and how an enhanced user model can enable sharing of ingested data. We describe how to make this component fault-tolerant so the system manages input in the presence of failures. We also show how to make this component elastic so that variances in incoming data rates can be handled gracefully without data loss if/when desired. Results from initial experiments that evaluate scalability and fault-tolerance of AsterixDB data feeds facility are reported. We include an evaluation of built-in ingestion policies and study their effect as well on throughput and latency. An evaluation and comparison with a 'glued' together system formed from popular engines — Storm (for streaming) and MongoDB (for persistence) — is also included.

1. INTRODUCTION

A large volume of data is being generated on a "continuous" basis, be it in the form of click-streams, output from sensors or via sharing on popular social websites [3]. Encouraged by low storage costs, enterprises today are aiming to collect and persist the available data and analyze it over time to extract useful information. Marketing departments use Twitter feeds to conduct sentiment analysis to get end user feedback on their company's products. As another example, utility companies have rolled out meters that measure the consumption of water, gas, and electricity and generate huge volumes of interval data that is analyzed over time. Traditional data management systems require data to be loaded and indexes to be created before data can be subjected to ad hoc analytical queries. To keep pace with "fast-moving" data, a Big Data Management System (BDMS) must be able to ingest and persist data on a continuous basis. A flow of data from an external source into persistent (indexed) storage inside a BDMS will be referred to here as a *data feed*. The task of maintaining the continuous flow of data is hereafter referred to as *data feed management*.

A simple way of having data being put into a Big Data management system on a continuous basis is to have a single program (process) fetch data from an external data source, parse the data, and then invoke an insert statement per record or batch of records. This solution is limited to a single machine's computing capacity. Ingesting multiple data feeds would potentially require running and managing many individual programs/processes. The task of continuously retrieving data from external source(s), applying some pre-processing for cleansing, filtering data, and indexing the processed data today amounts to 'gluing' together different systems (e.g. [19]). It becomes hard to reason about the data consistency, scalability and fault-tolerance offered by such an assembly. Traditional data management systems have evolved over time to provide native support for services if the service offered by an external system is inappropriate or may cause substantial overheads [18, 10]. Responding to the new need then, it is natural for a BDMS to provide "native" support for data feed management.

1.1 Challenges in Data Feed Management

Let us begin by enumerating the key challenges involved in building a data feed facility.

C1) *Genericity* and *Extensibility*: A feed ingestion facility must be generic enough to work with a variety of data sources and high-level applications. A plug-and-play model is desired to allow extension of the offered functionality.

C2) *Fetch-Once, Compute-Many*: Multiple applications may wish to consume the ingested data and may wish the arriving data to be processed/persisted differently. It is desirable to receive a single flow of data from an external source and yet transform it in multiple ways to drive different applications concurrently.

C3) *Scalability and Elasticity*: Multiple feeds with fluctuating data arrival rates, coupled with ad hoc queries over the persisted data, imply a varying demand for resources. The system should offer *scalability* by being able to ingest increasingly large volumes of data (possibly from multiple sources) via the addition of resources. The system should demonstrate *elasticity* by auto-scaling in/out to meet the demand for resources.

C4) *Fault Tolerance*: Data ingestion is expected to run on a large cluster of commodity hardware that may be prone to hardware failures. It is desirable to offer the desired degree of robustness in handling failures while minimizing data loss.

10.5441/002/edbt.2015.61

1.2 Contributions

In this paper, we describe the support for data feed management in AsterixDB. AsterixDB provides a platform for the scalable storage and analysis of very large volumes of semi-structured data. The paper describes the approach adopted to address the aforementioned challenges. This paper also demonstrates the efficiency/flexibility achieved in having native support for feed ingestion in AsterixDB in comparison to the popular approach of 'gluing' together popular systems (e.g. Storm[6] and MongoDB[5]), which is the state of the art today. The paper offers the following contributions.

(1) *Concepts involved in Data Feed Management*: The paper introduces the concepts involved in defining a data feed and managing the flow of data into a target dataset and/or to other dependent feeds to form a cascade network. It details the design and implementation of the involved concepts in a complete system.

(2) *Policies for Data Feed Management*: We describe how a data feed is managed by associating an ingestion policy that controls the system's runtime behavior in response to failures and resource bottlenecks. Users may also opt to provide a custom policy to suit special application requirements.

(3) *Scalable/Elastic Data Feed Management*: We describe a dataflow approach that exploits partitioned-parallelism to scale and ingest increasingly large amounts of data. The dataflow exhibits elasticity by being able to monitor and dynamically re-structure itself to adapt to the rate of arrival of data. The system is fault-tolerant and provides *at least once semantics* as the strongest guarantee, if required.

(4) *Contribution to Open-Source*: AsterixDB is available as open source software [2, 1]. The support for data ingestion in AsterixDB is extensible to enable future contributors to provide custom implementations of different modules.

(5) *Experimental Evaluation*: We describe an experimental evaluation that studies the role of different ingestion policies in determining the behavioral aspects of the system including the achieved throughput and latency. We also report on experiments to evaluate scalability and our approach to fault-tolerance.

(6) *Improvement over State-of-the-Art*: We include an evaluation of a system created by coupling Storm (a popular streaming engine) and MongoDB (a popular persistence store) to draw a comparison with AsterixDB in terms of flexibility and scalability achieved in data feed management. We demonstrate and describe the inefficiencies involved in 'gluing' together such otherwise efficient systems; doing so is a current common practice in open-source community.

The rest of the paper is organized as follows. We discuss related work in Section 2 and provide an overview of AsterixDB in Section 3. Section 4 describes how a feed is modeled and defined at the language level in AsterixDB. The implementation details are described in Section 5. Section 6 describes the support for handling failures. Section 7 provides an experimental evaluation, and we conclude in Section 8.

2. RELATED WORK

Data feeds may seem similar to streams from the data streams literature (e.g. [7, 13]). There are important differences, however. Data feeds are a "plumbing" concept; they are a mechanism for having data flow from external sources that produce data continuously to incrementally populate and persist the data in a data store. Stream Processing Engines (SPEs) do not persist data; instead they operate with a sliding window on data (e.g. a 5 minute view of data), but the amount, or the time window, is usually limited by the velocity of the data and the available memory. In a similar spirit, Complex Event Processing (CEP) systems (Storm [6] and S4 [15]) can route, transform and analyze a stream of data. However, these systems do not persist the data or provide support for ad hoc analytical queries. These engines can be used in conjunction with a database (e.g MySql or MongoDB), making it possible to persist and run ad hoc queries.

In the past, ETL (Extract Transform Load) systems (e.g. [4]) have supported populating a Data Warehouse with data collected from multiple data sources. However, such systems operate in a "batchy" mode, with a "finite" amount of data transferred at periodic intervals coinciding with off-peak hours. Xu et al. in [19] described a Map-Reduce based approach for populating a parallel database system with an external feed. However, the system was tightly-coupled with Map-Reduce and required data to be put into HDFS, thus involving an additional copy.

With respect to providing fault-tolerance, stream processing systems also faced the challenge of providing highly available parallel data-flows and have proposed several techniques [17, 8]. The process-pairs approach, used in Flux and StreamBase, involves a high overhead when the system needs to scale. These techniques rely on replication; the state of an operator is replicated on multiple servers or have multiple servers simultaneously process identical input streams. Fault-tolerance is provided at a high cost, as the number of nodes is thus at least doubled due to replication. It was thus not considered for use in AsterixDB. Moreover, offering a single strong strategy for fault-tolerance can be wasteful of resources in scenarios where the offered degree of robustness exceeds the application's requirements.

Data ingestion and stream-processing data-flows are typically associated with fluctuating data arrival rates that cause a varying demand for resources. An elastic behavior with the ability to scale in/out in adaptation to the demand for resources is desirable. Such mechanisms have been studied and evaluated before. Elastic reconfiguration in [16] is triggered when the data arrival rate exceeds a pre-calculated saturation rate by some fraction (e.g., 5% in their papers). It is not clear how an appropriate saturation rate would be statically calculated in a dynamic environment with concurrent feeds and queries over the ingested data. AsterixDB follows a different approach by dynamically monitoring the rate of flow of data and the availability of resources across the participant nodes. This allows detecting resource bottlenecks and triggering corrective actions in accordance with measured values.

We have explored the challenges involved in building a data ingestion facility. In doing so, we added a new BDMS architectural component, called a *data feed*, that makes a Big Data system the caretaker for functionality that used to live outside, and we show how it improves users' lives and system performance. In this paper, we describe how to build this component, architecturally, so that it provides continuous load-like performance (i.e., low overhead) - and how an enhanced user model can enable sharing of ingested data. We identify a number of different QoS options that users might want, depending on the nature of their application, and we show how to deliver them via dynamic monitoring of the system state. We also show how to make this new *data feed* component fault-tolerant so the system manages input (and the user doesn't have to) in the presence of failures. We show how to make this component elastic so that variances in incoming data rates can be handled gracefully without data loss if/when desired. We added that functionality to AsterixDB, and we demonstrate that it works (and how well).

3. BACKGROUND: ASTERIXDB

Initiated in 2009, the AsterixDB project has been developing new technologies for ingesting, storing, indexing, querying, and analyzing vast quantities of semi-structured data. It combines the ideas from three distinct areas—semi-structured data, parallel databases, and data-intensive computing—in order to create an open-source software platform that scales by running on large, shared-nothing commodity computing clusters.

3.1 AsterixDB Architecture

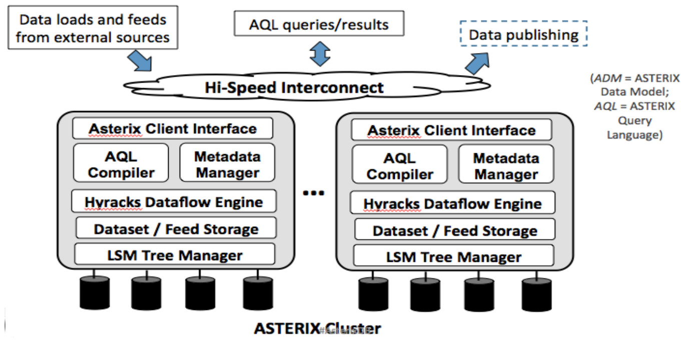


Figure 1: AsterixDB Architecture

Figure 1 provides an overview of how the various software components of AsterixDB map to nodes in a shared-nothing cluster. The topmost layer of AsterixDB is a parallel DBMS, with a full, flexible AsterixDB Data Model (ADM) and AsterixDB Query Language (AQL) for describing, querying, and analyzing data. ADM and AQL support both native storage and indexing of data as well as analysis of external data (e.g., data in HDFS). The bottom-most layers from Figure 1 provide storage facilities for datasets, which can be targets of ingestion. These datasets are stored and managed by AsterixDB as partitioned LSM-based B+-trees with optional LSM-based secondary indexes. A detailed description of AsterixDB and results from experimental evaluation can be found in [12].

AsterixDB uses Hyracks [11] as its execution layer. Hyracks allows AsterixDB to express a computation as a DAG of data operators and connectors. Operators operate on partitions of input data and produce partitions of output data, while connectors repartition operators' outputs to make the newly produced partitions available at the consuming operators.

3.2 AsterixDB Data Model

AsterixDB defines its own data model (ADM) [9] designed to support semi-structured data with support for bags/lists and nested types. Figure 2 shows how ADM can be used to define a record type for modeling a raw tweet. *RawTweet* type is an open type, meaning that its instances will conform to its specification but can contain extra fields that vary per instance. Figure 2 also defines a *ProcessedTweet* type. A processed tweet replaces the nested user field inside a raw tweet with a primitive userId value and adds a nested collection of strings (referred topics) to each tweet. Derived attributes about the tweet (e.g. sentiment and language) are also included. The primitive location field types (location-lat, location-long) and send-time are expressed as their respective spatial (point) and temporal (datetime) datatypes. ADM also allows specifying optional fields with known types (e.g. location).

Data in AsterixDB is stored in *datasets*. Each record in a dataset conforms to the datatype associated with the dataset. Data is hash-partitioned (primary key) across a set of nodes that form the *node-group* for a dataset, which defaults to all nodes in an AsterixDB cluster. Figure 3 shows the AQL statements for creating a pair of datasets—*RawTweets* and *ProcessedTweets*. We create a secondary index on the location attribute of a processed tweet for more efficient retrieval of tweets on the basis of spatial location.

```
create type RawTweet              create type TwitterUser
as open {                         as open {
   tweetId: string,                  screen-name: string,
   user: TwitterUser,                lang: string,
   location-lat: double?,            friends_count: int32,
   location-long: double?,           statuses_count: int32,
   send-time: string,                name: string,
   message-text: string              followers_count: int32
};                                };

create type ProcessedTweet as open {
   tweetId: string,
   userId: string,
   location: point?,
   send-time: datetime,
   message-text: string,
   referred-topics: {{string}},
   sentiment: double,
   language: string
};
```

Figure 2: Defining datatypes

```
create dataset RawTweets(RawTweet) primary key tweetId;

create dataset ProcessedTweets(ProcessedTweet)
primary key tweetId;

create index locationIndex on
ProcessedTweets(location) type rtree;
```

Figure 3: Creating datasets and associated indexes

4. DATA FEED BASICS

AQL has built-in support for data feeds. In this section, we describe how an end-user may model a data feed and have its data be persisted/indexed into an AsterixDB dataset.

4.1 Collecting Data: Feed Adaptors

The functionality of establishing a connection with a data source and receiving, parsing and translating its data into ADM records (for storage inside AsterixDB) is contained in a *feed adaptor*. A feed adaptor is an implementation of an interface and its details are specific to a given data source. An adaptor may optionally be given parameters to configure its runtime behavior. Depending upon the data transfer protocol/APIs offered by the data source, a feed adaptor may operate in a *push* or a *pull* mode. Push mode involves just one initial request by the adaptor to the data source for setting up the connection. Once a connection is authorized, the data source "pushes" data to the adaptor without any subsequent requests by the adaptor. In contrast, when operating in a pull mode, the adaptor makes a separate request each time to receive data.

AsterixDB currently provides built-in adaptors for several popular data sources—Twitter, CNN, and RSS feeds. AsterixDB additionally provides a generic socket-based adaptor that can be used to ingest data that is directed at a prescribed socket. Figure 4 illustrates the use of built-in adaptors in AsterixDB to define a pair of feeds. The *TwitterFeed* contains tweets that contain the word "Obama". As configured, the adaptor will make a request for data every minute. The *CNNFeed* will consist of news articles that are related to any of the topics that are specified as part of the indicated configuration.

```
create feed TwitterFeed using TwitterAdaptor
("api"="pull", "query"="Obama", "interval"=60);

create feed CNNFeed using CNNAdaptor
("topics"="politics, sports");
```

Figure 4: Defining a feed using some of the built-in adaptors in AsterixDB

The degree of parallelism in receiving data from an external source is determined by the feed adaptor in accordance with the data exchange protocol offered by the data source. The external source may allow transfer of data in parallel across multiple channels. For example, CNN as a data source offers an RSS feed corresponding to each topic (politics, sports, etc). The CNNFeed can thus employ a degree of parallelism as determined by the number of topics that are passed as configuration. Multiple instances will then run as parallel threads on a single machine or on multiple machines. In contrast, the TwitterAdaptor uses a single degree of parallelism.

4.2 Pre-Processing Collected Data

A feed definition may optionally include the specification of a user-defined function that is to be applied to each feed record prior to persistence. Examples of pre-processing might include adding attributes, filtering out records, sampling, sentiment analysis, feature extraction, etc. The pre-processing is expressed as a user-defined function (UDF) that can be defined in AQL or in a programming language like Java. An AQL UDF is a good fit when pre-processing a record requires the result of a query (join or aggregate) over data contained in AsterixDB datasets. More sophisticated processing such as sentiment analysis of text is better handled by providing a Java UDF. A Java UDF has an initialization phase that allows the UDF to access any resources it may need to initialize itself prior to being used in a data flow. It is assumed by the AsterixDB compiler to be stateless and thus usable as an embarassingly parallel black box. In constrast, the AsterixDB compiler can reason about an AQL UDF and involve the use of indexes during its invocation.

The tweets collected by the TwitterAdaptor (Figure 4) conform to the *RawTweet* datatype (Figure 2). The processing required in transforming a collected tweet to its lighter version (of type *ProcessedTweet*) involves extracting the hash tags (if any) in a tweet and collecting them in the referred-topics attribute for the tweet. Attributes associated with a tweet (sentiment, language) are derived from analyzing the text. In the case of the *CNNFeed*, the CNNAdaptor (Figure 4) outputs records that each contain the fields item, link and description. The *link* field provides the URL of the news article on the CNN website. Parsing the HTML source provides additional information such as tags, images and outgoing links to other related articles. This information can then be added to each record as additional fields prior to persistence. The pre-processing function for a feed is specified using the *apply function* clause at the time of creating the feed (Figure 5).

A feed adaptor and a UDF act as *pluggable* components. These contribute towards providing a generic 'plug-and-play' model where custom implementations can be provided to cater to specific requirements. This helps address challenge C1 from Section 1.1. By providing implementation of the prescribed interfaces, the internal details of data feed management are abstracted from end users. A feed adaptor or a Java UDF can be packaged and installed as part of an AsterixDB library and subsequently be used in AQL statements. A tutorial on building a custom feed adaptor or a UDF with a description of the interfaces to be implement can be found at [1].

```
create feed ProcessedTwitterFeed using TwitterAdaptor
("api"="pull", "query"="Obama", "interval"=60)
apply function addFeatures;

create feed ProcessedCNNFeed using CNNAdaptor
("topics"="politics, sports")
apply function addInfoFromCNNWebsite;
```

Figure 5: Defining a feed that involves pre-processing of collected data

4.3 Building a Cascade Network of Feeds

Multiple high-level applications may wish to consume the data ingested from a data feed. Each such application might perceive the feed in a different way and require the arriving data to be processed and/or persisted differently. Building a separate flow of data from the external source for each application is wasteful of resources as the pre-processing or transformations required by each application might overlap and could be done together in an incremental fashion to avoid redundancy. A single flow of data from the external source could provide data for multiple applications. To achieve this, we introduce the notion of *primary* and *secondary* feeds in AsterixDB to address challenge C2 from Section 1.1.

A feed in AsterixDB is considered to be a *primary* feed if it gets its data from an external data source. The records contained in a feed (subsequent to any pre-processing) are directed to a designated AsterixDB dataset. Alternatively or additionally, these records can be used to derive other feeds known as *secondary* feeds. A secondary feed is similar to its parent feed in every other aspect; it can have an associated UDF to allow for any subsequent processing, can be persisted into a dataset, and/or can be made to derive other secondary feeds to form a *cascade network*. A primary feed and a dependent secondary feed form a hierarchy. As an example, Figure 6 shows the AQL statements that redefine the previous feeds—*ProcessedTwitterFeed* and *ProcessedCNNFeed*—in terms of their respective parent feeds from Figure 4.

```
create secondary feed ProcessedTwitterFeed from
feed TwitterFeed apply function addFeatures;

create secondary feed ProcessedCNNFeed from
feed CNNFeed apply function addInfoFromCNNWebsite;
```

Figure 6: Defining a secondary feed

4.4 Lifecycle of a Feed

A feed is a *logical* artifact that is brought to life (i.e. its data flow is initiated) *only* when it is *connected* to a dataset using the *connect feed* AQL statement (Figure 7). Subsequent to a connect feed statement, the feed is said to be in the *connected* state. Multiple feeds can simultaneously be connected to a dataset such that the contents of the dataset represent the union of the connected feeds. In a supported but unlikely scenario, one feed may also be simultaneously connected to different target datasets. Note that connecting a secondary feed does not require the parent feed (or any ancestor feed) to be in the *connected* state; the order in which feeds are connected to their respective datasets is not important. Furthermore, additional (secondary) feeds can be added to an existing hierarchy and connected to a dataset at any time without impeding/interrupting the flow of data along a connected ancestor feed.

```
connect feed ProcessedTwitterFeed to
dataset ProcessedTweets;

disconnect feed ProcessedTwitterFeed from
dataset ProcessedTweets;
```

Figure 7: Managing the lifecycle of a feed

The *connect feed* statement in Figure 7 directs AsterixDB to persist the *ProcessedTwitterFeed* feed in the *ProcessedTweets* dataset. If it is required (by the high-level application) to also retain the raw tweets obtained from Twitter, the end user may additionally choose to connect *TwitterFeed* to a (different) dataset. Having a set of primary and secondary feeds offers the flexibility to do so. Let us assume that the application needs to persist *TwitterFeed* and that, to do so, the end user makes another use of the *connect feed* statement. A logical view of the continuous flow of data established by connecting the feeds to their respective target datasets is shown in Figure 8. The flow of data from a feed into a dataset can be terminated explicitly by use of the *disconnect feed* statement (Figure 7). Disconnecting a feed from a particular dataset does not interrupt the flow of data from the feed to any other dataset(s), nor does it impact other connected feeds in the lineage.

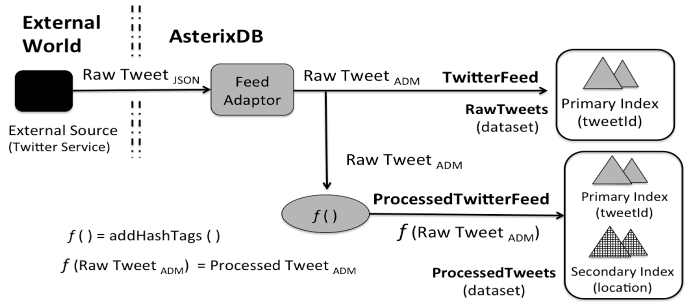


Figure 8: Logical view of the flow of data from external data source into AsterixDB datasets

4.5 Policies for Feed Ingestion

Multiple feeds may be concurrently operational on an AsterixDB cluster, each competing for resources (CPU cycles, network bandwidth, disk IO) to maintain pace with their respective data sources. A data management system must be able to manage a set of concurrent feeds and make dynamic decisions related to the allocation of resources, resolving resource bottlenecks and the handling of failures. Each feed has its own set of constraints, influenced largely by the nature of its data source and the application(s) that intend to consume and process the ingested data. Consider an application that intends to discover the trending topics on Twitter by analyzing the *ProcessedTwitterFeed* feed. Losing a few tweets may be acceptable. In contrast, when ingesting from a data source that provides a click-stream of ad clicks, losing data would translate to a loss of revenue for an application that tracks revenue by charging advertisers per click.

AsterixDB allows a data feed to have an associated *ingestion policy* that is expressed as a collection of parameters and associated values. An ingestion policy dictates the runtime behavior of the feed in response to resource bottlenecks and failures. Note that during push-based feed ingestion, data continues to arrive from the data source at its regular rate. In a resource-constrained environment, a feed ingestion framework may not be able to process and persist the arriving data at the rate of its arrival. AsterixDB provides a list of policy parameters (Table 1) that help customize the system's runtime behavior when handling excess records. AsterixDB provides a set of built-in policies, each constructed by setting appropriate value(s) for the policy parameter(s) from Table 1.

The handling of excess records by the built-in ingestion policies of AsterixDB is summarized in Table 2. Buffering of excess records in memory under the 'Basic' policy has clear limitations given that memory is bounded and may result in a termination of the feed if the available memory or the allocated budget is exhausted. The 'Spill' policy resorts to spilling the excess records to the local disk for deferred processing until resources become available again. Spilling is done intermittently during ingestion when required, and the spillage is processed as soon as resources (memory) are available. In contrast, the 'Discard' policy causes the excess records to be discarded altogether until the existing backlog is cleared. However, this results in periods of discontinuity when no records received from the data source are persisted. This behavior may not be acceptable to an application wishing to consume the ingested data. A best-effort alternative is provided by the 'Throttling' policy, wherein records are randomly filtered out (sampled) to effectively reduce their rate of arrival. In addition, AsterixDB also provides the 'Elastic' policy, which attempts to scale-out/in by increasing/decreasing the degree of parallelism involved in processing of records. We will discuss the built-in policies to Section 5.3, where we cover the physical aspects and their implementation details.

Note that the end user may choose to form a custom policy. E.g. it is possible in AsterixDB to create a custom policy that spills excess records to disk and subsequently resorts to throttling if the spillage crosses a configured threshold. In all cases, the desired ingestion policy is specified as part of the connect feed statement (Figure 9) or else the 'Basic' policy will be chosen as the default. It is worth noting that a feed can be connected to a dataset at any time, which is independent from other related feeds in the hierarchy. As such the *connect feed* statements shown in Figure 9 are not required to be executed together.

The ability to form a custom policy allows the runtime behavior to customized as per the specific needs of the high-level application(s) and helps address challenge C1 from Section 1.1.

```
connect feed TwitterFeed to dataset RawTweets
using policy Basic;

connect feed ProcessedTwitterFeed to
dataset ProcessedTweets using policy Basic;
```

Figure 9: Specifying the ingestion policy for a feed

5. RUNTIME FOR FEED INGESTION

So far we have described, at a logical level, the user model and built-in support in AQL that enables the end user to define a feed,

Table 1: A Few Important Policy Parameters

Policy Parameter	Description	Default
excess.records.spill	Set to true if records that cannot be processed by an operator for lack of resources (referred to as *excess records* hereafter) should be persisted to the local disk for deferred processing.	false
excess.records.discard	Set to true if *excess records* should be discarded.	false
excess.records.throttle	Set to true if rate of arrival of records is required to be reduced in an adaptive manner to prevent having any *excess records*.	false
excess.records.elastic	Set to true if the system should attempt to resolve resource bottlenecks by re-structuring and/or rescheduling the feed ingestion pipeline.	false
recover.soft.failure	Set to true if the feed must attempt to survive any runtime exception. A false value permits an early termination of a feed in such an event.	true
recover.hard.failure	Set to true if the feed must attempt to survive a hardware failures (loss of AsterixDB node(s)). A false value permits the early termination of a feed in the event of a hardware failure.	true

manage its lifecycle, and dictate its runtime behavior by selecting a policy. Next, we discuss the physical aspects and implementation details involved in building and managing the flow of data when a feed is connected to a dataset.

5.1 Runtime Components

In processing a connect feed statement, the AQL compiler retrieves the definitions of the involved components—feed, adaptor, function, policy, and the target dataset from the AsterixDB Metadata. The compiler translates a *connect feed* statement into a Hyracks job that is subsequently scheduled to run on an AsterixDB cluster. The resulting dataflow is referred to as a feed ingestion pipeline. A Hyracks *data operator* forms a major building block of an ingestion pipeline and is useful in executing custom logic on partitions of input data to produce partitions of output data. It may employ parallelism in consuming input by having multiple instances that run in parallel across a set of nodes in an AsterixDB cluster. *Data connectors* repartition operators' outputs to make the newly produced partitions available at the consuming operator instances. In addition, an ingestion pipeline provides *feed joints* at specific locations. A *feed joint* is like a network tap and provides access to the data flowing along an ingestion pipeline. It helps in building a cascade network of feeds by allowing data from an ingestion pipeline to be simultaneously routed along multiple paths.

A feed ingestion pipeline involves 3 stages—*intake*, *compute* and *store*. The *intake* stage involves creating an instance of the associated feed adaptor, using it to initiate the transfer of data and transforming it into ADM records. If the feed has an associated preprocessing function, it is applied to each feed record as part of the

Table 2: Policies for handling of excess records

Policy	Approach to handling of excess records
Basic	Buffer excess records in memory
Spill	Spill excess records to disk for deferred processing
Discard	Discard excess records altogether
Throttle	Randomly filter out records to regulate the rate of arrival
Elastic	Scale out/in to adapt to the rate of arrival

compute stage. Subsequently, as part of the *store* stage, the output records from the preceding *intake* or a *compute* stage are put into the target dataset and its secondary indexes[1] (if any) are updated accordingly. Each stage is handled by a specific data-operator, hereafter referred to as an *intake*, *compute*, and *store* operator respectively. Next, we describe how operators, connectors and joints are assembled together to construct a feed ingestion pipeline.

Figure 10 contains some example AQL statements that define and connect a pair of feeds to respective target datasets. The second statement in Figure 10 connects the primary feed, CNNFeed. As determined by the number of topics specified in its configuration, the feed involves the use of a pair of instances of the CNNFeedAdaptor. Each adaptor instance is managed by an instance of the *intake* data operator. As *CNNFeed* does not involve any preprocessing, the output records from each adaptor instance thus constitute the feed. These are then partitioned across a set of store operator instances using the hash-partitioning data-connector. In the constructed pipeline, a feed joint is located at the output of each intake operator instance. In general, a feed joint is placed at the output side of an operator instance that produces records that form a feed. In the case where a feed involves pre-processing, a feed joint is placed at the output of its *compute* operator instances.

The last statement in Figure 10 connects the secondary feed, *ProcessedCNNFeed*. By definition, this feed can be obtained by subjecting each record from the *CNNFeed* to the associated UDF (*addInfoFromCNNWebsite*). In general, if $feed_{m+1}$ denotes the immediate child of $feed_m$, a child feed $feed_i$ can be obtained from an ancestor feed $feed_k$ $(k < i)$ by subjecting each record from $feed_k$ to the sequence of UDFs associated with each child feed $feed_j$ $(j = k + 1, ..., i)$. $i - k$ denotes the 'distance' from $feed_k$ to $feed_i$ and is indicative of the additional processing steps (UDFs) required to produce $feed_i$ from $feed_k$. To minimize the processing involved in forming a feed, it is desired to source the feed from its nearest ancestor feed that is in the connected state. The feed joint(s) available along the ingestion pipeline of an ancestor feed are then used to access the flowing data and subject it to the additional processing to form the desired feed. AsterixDB keeps track of the available feed joints and uses them in preference over creating a new feed adaptor instance in sourcing a feed.

The cascade network for ingestion of *CNNFeed* and *ProcessedCNNFeed* is shown in Figure 11. Note that disconnecting a feed from a dataset does not necessarily remove the set of feed joints located along the ingestion pipeline. Referring to Figure 11, disconnecting CNNFeed at this stage removes the tail of the pipeline that includes the compute and store operator instances but will retain the intake operator instances. This is because the feed joints (labeled as 'A') at the output of the intake operator instances each have an existing

[1]Secondary indexes in AsterixDB are partitioned and co-located with the corresponding primary index partition. Inserting a record into the primary and any secondary indexes uses write-ahead logging and offers ACID semantics.

path (ingestion pipeline for ProcessedCNNFeed) that requires the output records to keep flowing in an uninterrupted manner.

```
create feed CNNFeed using CNNAdaptor
("topics"="politics, sports");

connect feed CNNFeed to dataset RawArticles;

create secondary feed ProcessedCNNFeed from
feed CNNFeed apply function addInfoFromCNNWebsite;

connect feed ProcessedCNNFeed to
dataset ProcessedArticles;
```

Figure 10: Example AQL statements

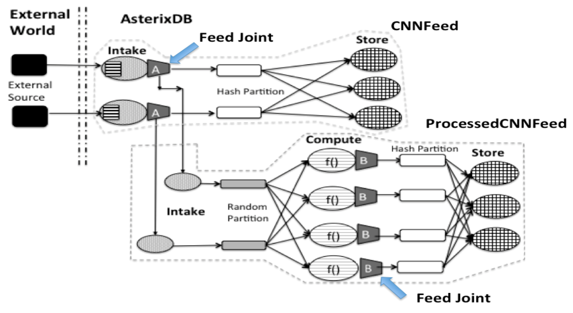


Figure 11: An example of a feed cascade network. The cascade network provides two sets of feed joints - labeled as 'A' & 'B' - that provide access to CNNFeed and CNNProcessedFeed respectively. If at this stage, the end-user creates (and connects) a secondary feed that derives from *ProcessedCNNFeed*, then its intake stage would involve receiving records from each of the 4 feed joints (kind B) provided by the ingestion pipeline for *ProcessedCNNFeed*.

5.2 Scheduling a Feed Ingestion Pipeline

Scheduling a feed ingestion pipeline on a cluster requires determining the desired degree of parallelism of each operator and mapping each instance of an operator to an AsterixDB node. An AsterixDB cluster consists of a manager node and a set of worker nodes. Scheduling decisions for a feed ingestion pipeline are taken by the *Central Feed Manager* (CFM) that is hosted by the manager node. Each worker node hosts a *Feed Manager* (FM). The CFM keeps track of the load distribution across the cluster from the periodic reports sent by each FM. These reports contain vital statistics including CPU usage. Each feed pipeline operator may define a cardinality constraint (degree of parallelism) and/or a location constraint at the Hyracks job level. The location and cardinality constraints for the intake operator are determined by the feed adaptor. If no constraints are specified, the Central Feed Manager will choose to run a single instance of the operator on a node of its choice. The constraints for the store operator are pre-determined and derived from the nodegroup associated with the target dataset. Recall that the nodegroup of a dataset refers to the set of nodes that hold the partitions of the dataset.

The compute operator in a feed pipeline doesn't offer any constraints. Instead, the level of parallelism for a compute operator is determined as described below. The partitioned parallelism employed at the compute and store stages helps the system ingest increasingly large volumes of data. Additional resources (physical machines) can be added at the compute and/or store stage to scale out the system. This helps address challenge C5 from Section 1.1. The appropriate degree of parallelism is therefore dependent on the rate of arrival of data and on the complexity associated with the UDF. To begin with, the cardinality at the compute stage is matched with that of the store stage to offer the same degree of parallelism. However, as we describe in the following section, a feed ingestion pipeline, as dictated by the ingestion policy, may be re-structured in accordance with the demand for resources.

5.3 Managing Congestion

An expensive UDF and/or an increased rate of arrival of data may lead to an excessive demand for resources leading to delays in the processing of records. Left unchecked, the created *back-pressure* at an operator can cascade upstream to completely 'lock' the flow of data along the pipeline. A 'locked' state creates excessive demand for resources to buffer the data that is arriving at its regular rate from the data source(s). At a feed joint located on a 'locked' pipeline, insufficient resources can also impede the flow of data along other pipelines originating from the joint. To avoid such an undesirable situation, AsterixDB takes a different approach by resolving back-pressure at its originating operator and preventing it from escalating upstream. This isolates other operators in the pipeline/cascade network from the created congestion. In this sub-section, we describe the methodology adopted for detecting resource bottlenecks and taking corrective action (challenge C3 from Section 1.1).

The records arriving at an operator are buffered in memory; this functionality is provided by a *MetaFeed* operator that wraps around the actual operator (referred to as the *core* operator hereafter). Having a MetaFeed operator as a wrapper ensures that the core operators remain simple, generic and reusable elsewhere as part of other (non-feed) jobs. In addition to buffering, the MetaFeed operator periodically measures the size of the input buffer, the rate of arrival of records R_A (records/sec), and the rate of processing of records R_P (records/sec) by the core operator. Note that R_P varies with the availability of resources at the operator location and the size of the records that need to be processed. If R_A exceeds R_P, the buffer is expanded to accommodate for the deficit. The total available memory (JVM heap size) is bounded and is shared by operators serving multiple concurrent feeds or queries. To ensure sufficient resources for concurrent queries, a fixed (configurable) limit is imposed on the total memory allocated for feed input buffers at each worker node.

Table 1 from Section 4.5 described buffering, spilling, discarding or throttling as mechanisms for dealing with congestion. These mechanisms constitute 'local' resolution and remain hidden from the upstream operators that continue to send data seamlessly. The mechanisms' downside is delayed processing of records (buffering/spilling) or losing some of them altogether (discarding/throttling).

Congestion that occurs due to a compute operator (i.e., due to use of an expensive UDF) can be cleared in yet another way – via 'global' resolution. Global resolution exploits the stateless and therefore embarrassingly parallel nature of the UDF. The MetaFeed operator reports a *congested* state of a compute operator to the local Feed Manager (FM) together with the last measured values for R_A and R_P. Congested states occurring across the cluster are reported to the Central Feed Manager (CFM) by each FM. Using mechanisms similar to those detailed later for handling failures during ingestion, the CFM re-structures the pipeline to have increased parallelism at the compute tier. In doing so, the additional compute operator instances may run on idle nodes from the cluster or be scheduled on the current set of nodes to utilize additional cores. Contrary to dynamically scaling out an operator, AsterixDB also provides for auto-scaling-in if the current degree of parallelism is greater than that required to handle the flow of data. The required increase/decrease is derived from the reported values of R_A and R_P from the compute operator instances running across the cluster.

5.4 At Least Once Semantics

An application may demand stronger guarantees on the processing of records by requiring each arriving record to be processed *at least once* through the ingestion pipeline, despite any failures. Such a requirement is expressed through the *at.least.once.enabled* policy parameter. To provide *at least once semantics*, each record arriving from the data source is augmented with a *tracking id* at the intake stage. Once the record has been persisted (log record on disk), an *ack* message with the *tracking id* is constructed. Over a fixed-width time-window, the ack messages for all records that were sourced from a given feed adaptor instance (identified from the tracking id) are grouped and encoded together as a single message. A record that has been output by the intake stage is held at its intake node until an ack message for the record is received from the store stage. When an ack is received, the record is dropped and memory is reclaimed. On a timeout, the records without an ack are replayed. *At least once* semantics are not guaranteed if *throttling* or *discarding* of records is enabled by the policy.

6. FAULT TOLERANT FEED INGESTION

Feed ingestion is a long running task running on a commodity cluster, so it is eventually bound to encounter hardware failure(s). Also, portions of a feed ingestion pipeline include pluggable user-provided modules (feed adaptor and a pre-processing function) that may cause soft failures (runtime exceptions). Sources of such an exception may include unexpected data format/values or simply inherent bugs in the user-provided source code. Next, we describe how a feed may recover from software and hardware failures and thereby address the challenge C4 from Section 1.1.

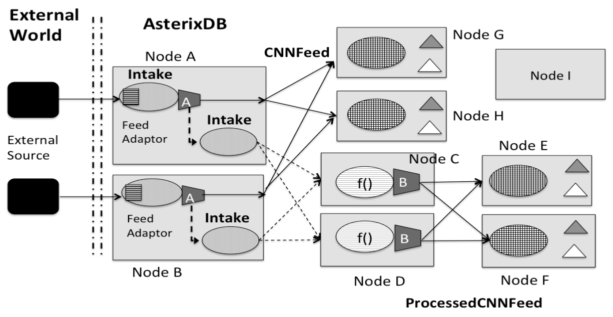
6.1 Handling Software Failures

A runtime exception encountered by an operator in processing an input record in a normal AsterixDB insert setting carries non-resumable semantics and causes the insert operation's dataflow to cease. It is essential to guard feed pipelines from such exceptions by executing each of their operators in a sandbox-like environment. The MetaFeed operator (introduced in Section 5.3) acts as a shell around each operator to provide such an environment. Recall that the operator that is wrapped is referred to as the *core* operator. The runtime of a core operator receives input data as a sequence of frames containing one ore more records. An exception thrown by the core operator in processing an input record is caught by the wrapping MetaFeed operator. The MetaFeed operator slices the original input frame to form a subset frame that includes the unprocessed records minus the exception generating record. The subset frame is then passed to the core-operator which continues to process input frames and has, in effect, skipped past the exception-generating record.

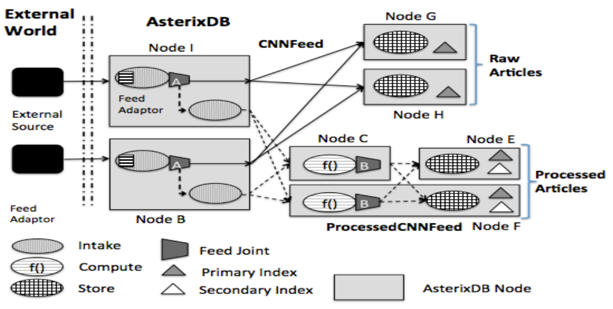
6.2 Handling Hardware Failures

We next describe the mechanism that handles the loss of one or more of the AsterixDB nodes involved in a feed ingestion pipeline. Corresponding to the operation being performed, a node is referred to as an *Intake*, *Compute* or a *Store* node. An AsterixDB cluster node may simultaneously act as an intake, compute or a store node for one or more feeds. To illustrate a failure scenario, we use an example ingestion pipeline (Figure 12(a)) that executes on a 10 member AsterixDB cluster (nodes A-I). In this particular data flow, node I is not used initially. To be considered *alive*, each node is required to send periodic heartbeats to the master node (not shown in the figure). We assume a concurrent failure of an intake node (node A) and a compute node (node D). A node failure is detected by the master node through the heartbeat mechanism. Each operator instance in the ingestion pipeline is notified of the pipeline failure. On being notified, the operator instance saves the contents of its input buffer with the local Feed Manager. The operator instance also has an option to save state information that may help in resuming operation once the pipeline is rescheduled. The operator instance then notifies the Central Feed Manager (CFM) and terminates itself. An *intake* operator instance behaves differently; it begins to buffer/spill the arriving records and not forward them downstream.

A revised feed ingestion pipeline is constructed with identical operators and feed joints. An operator instance is co-located with its respective instance from the previous failed execution if the node is still available. An intake operator instance is co-located with the corresponding *live* instance from the previous execution. Each operator then enters a 'hand-off' stage where it retrieves the input buffer and any state saved with the local Feed Manager by the corresponding instance from the failed pipeline. The functionality of registering with the Feed Manager and saving/retrieving any state across failures is provided by the *MetaFeed* operator that wraps around each core operator. An operator instance that has a *dead* instance from the previous execution can be scheduled to run at an AsterixDB node chosen by the CFM.



(a) An example dataflow for describing the fault tolerance protocol: Node A and Node D fail



(b) Restructured pipeline post recovery: Node I takes over Node A operations; Node F takes over Node D operations

Figure 12: Recovering from compute node failure.

When substituting a failed node, the CFM considers the load distribution across the cluster. Recall that the FMs periodically report vital statistics (including CPU usage) about a worker node to the

CFM. Figure 12(b) shows the revised pipeline with node I (which was idle) substituted for node A while Node F substituted for node D and thereafter acted as a compute and a store node. A more detailed description of the handling of different failure scenarios during feed ingestion can be found in [14]. Essentially the same machinery is used to handle the scaling-in or out of a feed pipeline when an elastic policy is chosen for handling data congestion (or decongestion) and that CFM determines that a scaling action is required.

A store node failure translates to the loss of a partition of the dataset that is receiving the feed. AsterixDB does not (yet!) support data replication. In the absence of replica(s), a store node failure will result in an *early* termination of an associated feed. When the failed store node later re-joins the cluster, it will undergo a log-based recovery to ensure that all of its hosted dataset partitions are in a consistent state. Subsequently, the feed ingestion pipeline will be rescheduled to involve the joined node.

7. EXPERIMENTAL EVALUATION

In this section, we provide an initial experimental evaluation of the scalability and fault-tolerance offered by the AsterixDB feed ingestion facility. We include a study of the impact of the ingestion policy (parameters) on the runtime behavior (throughput and latency) under different workload conditions. We also include a comparison with a custom built 'glued' system using Storm (as a processing engine) and MongoDB (as a persistence engine) and discuss the tradeoffs.

Experimental Setup: We ran experiments on a 10-node IBM x3650 cluster. Each node had one Intel 2.26GHz processor with four cores, 8GB of RAM, and a 300GB hard disk. We wrote a custom stand-alone tweet generator (*TweetGen*) that can output synthetic (JSON) tweets at a rate (*tweets per second* - ***twps***) that follows a configurable pattern. The *RawTweet* datatype created in Figure 2 showed the equivalent ADM representation for a tweet output by TweetGen. Next, we wrote a custom socket-based adaptor—*TweetGenAdaptor*. The adaptor is configured with the location(s) (socket address) where instance(s) of TweetGen is/are running. Each instance of TweetGen receives a request for data from a corresponding instance of *TweetGenAdaptor*, thus enabling ingestion of data in parallel. We used the AQL statements shown in Figure 3 (from Section 3.2) to create the target datasets (and indexes) for persisting the feed. The definition for the set of two feeds used in our experiment is shown in Figure 13. In this example, a pair of instances of TweetGen are running and listening on a specific for request to initiate (push-based) transfer of data.

```
create feed TweetGenFeed using TweetGenAdaptor
("datasource"="10.1.0.1:9000, 10.1.0.2:9000"));

create secondary feed ProcessedTweetGenFeed from
feed TweetGenFeed apply function addFeatures;
```

Figure 13: Feed definitions for experimental evaluation

7.1 Scalability

We evaluated the ability of the AsterixDB feed ingestion support to scale and ingest an increasingly large volume of data when additional resources are added. If the data arrival rate exceeds the rate at which it can be ingested in AsterixDB, the excess records are either buffered in memory, spilled to disk or discarded altogether. The precise behavior is chosen by the associated ingestion policy. By design, we chose to discard data here and not defer its processing (via spilling/buffering). This helped in evaluating the ability to successfully ingest data as a function of available resources.

In this experiment, we chose the amount of data loss as our performance metric. A total of 6 instances of TweetGen were run on machines outside the test cluster and were configured to generate tweets at a constant rate (20k twps) for a duration of 20 minutes. We measured the total number of ingested (persisted and indexed) tweets and repeated the experiment by varying the size of our test cluster. We increased the hardware till there is no data loss (the ideal behavior). The experimental results are shown in Figure 14. A significant proportion of records were discarded for lack of resources on a small size cluster of 1–4 nodes. On a bigger cluster, the proportion of discarded tweets declines, indicating that the system that can indeed ingest an increasingly high volume of data when additional resources (nodes) are added.

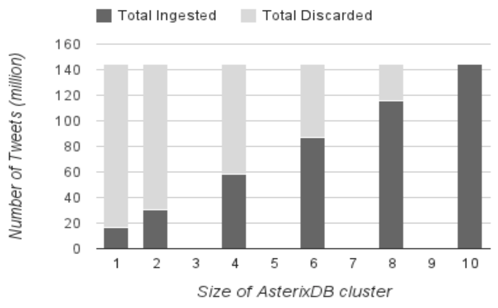


Figure 14: Scalability: Number of records (tweets) successfully ingested (persisted and indexed) as the cluster size is increased.

7.2 Fault Tolerance

We next evaluated the ability of the system to recover from single/multiple hardware failures while continuing to ingest data. This experiment involved a pair of TweetGen instances (twps=5000), each running on a separate machine outside the AsterixDB cluster. We connected the feeds—*TweetGenFeed* and *ProcessedTweetGenFeed*–to their respective target dataset and used the built-in policy *Fault-Tolerant* (Figure 15). The nodegroup associated with each dataset included a pair of nodes each. To make things interesting and show that the order of connecting related feeds is not important, we connected *ProcessedTweetGenFeed* prior to connecting its parent feed *TweetGenFeed*. In the absence of an available feed joint, the ingestion pipeline for *ProcessedTweetGenFeed* is constructed using the feed adaptor (Figure 16). The physical layout of the dataflow as scheduled on our AsterixDB cluster during this experiment is shown in Figure 16. The ingestion pipeline for *TweetGenFeed* is sourced from the feed joints (kind A) provided by *ProcessedTweetGenFeed*.

```
connect feed ProcessedTweetGenFeed to
dataset ProcessedTweets using policy FaultTolerant;

connect feed TweetGenFeed to
dataset RawTweets using policy FaultTolerant;
```

Figure 15: Connected feeds to respective dataset

We measured the number of records inserted into each target dataset during consecutive 2 second intervals to obtain the instantaneous ingestion throughput for the associated feed. We caused a compute node failure (node C in Figure 16) at t=70 seconds. This was followed by a concurrent failure of both an intake node (node

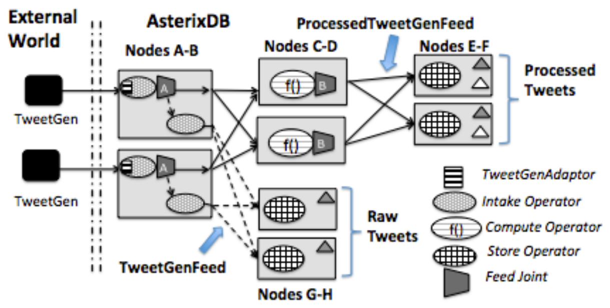
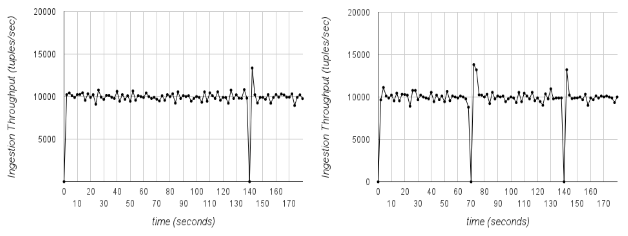


Figure 16: Feed cascade network for fault tolerance experiment: Node C fails at t=70 seconds; Node A and Node D fail at t=140 seconds



(a) TweetGenFeed (b) ProcessedTweetGenFeed

Figure 17: Instantaneous ingestion throughput with interim hardware failures: in Figure 16 Node C fails at t=70 seconds; Node A and Node D fail at t=140 seconds

A) and a compute node (node D) at t=140 seconds. The instantaneous ingestion throughput for each feed as plotted on a timeline is shown in Figure 17. Following are the noteworthy observations.

(i) **Recovery Time**: The failures are reflected as a drop in the instantaneous ingestion throughput at the respective times. Each failure was followed by a recovery phase that reconstructed the ingestion pipeline and resumed the flow of data into the target dataset (within 2-4 seconds).

(i) **Fault Isolation**: Data continues to arrive from the external source at the regular rate, irrespective of any failures in an AsterixDB cluster. During the recovery phase for *ProcessedTweetGenFeed*, the feed joint(s) buffer the records until the pipeline is resurrected but allow the records to flow (at their regular rate) into any other ingestion pipeline that does not involve a failed node. This helps in "**localizing**" the impact of a pipeline failure and is a desirable feature of the system. As shown in Figure 17(a), *TweetGenFeed* is not impacted by the failure of node C at t = 70 seconds.

7.3 Throughput and Latency: Impact of Ingestion Policy

We evaluated the impact of the ingestion policy on the runtime behavior and performance characteristics of interest — *throughput* and *ingestion latency* — under different conditions of rate of arrival of data. We configured two instances of TweetGen to generate tweets at a rate that followed the pattern shown in Figure 18(a). The pattern involves equi-width workload-phases with mid, high and low activity in terms of the rate of arrival of tweets. These workload-phases are referred to as W_{MID}, W_{HIGH} and W_{LOW} respectively and the corresponding rate (tweets/second or twps) is denoted by R_{MID}, R_{HIGH} and R_{LOW}. The pre-processing of tweets involved a UDF that simply executed a busy spin loop to consume CPU cycles and cause a compute delay of about 3ms per tweet.

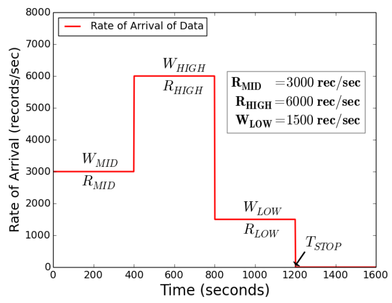
We used our 10 node AsterixDB cluster. Table 3 lists the symbols and metrics we use when describing this experiment and its results. The intake stage involved a pair of feed adaptor instances each receiving records from a separate TweetGen instance located outside the AsterixDB cluster. Each TweetGen instance pushed data for a continuous duration of 1200 seconds ($T_{start} - T_{stop}$). We measured the *instantaneous throughput* as the number of tweets persisted in each 2 second interval over the duration of the experiment. We also measured the *ingestion latency* (Table 3(b)) for each tweet received by the feed adaptor during each workload-phase (W_{MID}, W_{HIGH} and W_{LOW}). The target dataset had a partition on a disk at each node. The store stage thus involved a store operator instance on each node. The compute stage (as constructed by the AsterixDB compiler) offered a similar degree of parallelism and involved a compute operator instance on each node. Application of the UDF (with its ∼3ms execution time) by a compute operator instance gave each one a maximum processing capacity of ∼300 tweets/sec. The aggregate capacity from 10 parallel instances was thus limited to 3000 twps (referred to as $\text{Compute}_{\text{LIMIT}}$). In the workload (Figure 18(a)), we have $R_{\text{HIGH}} > \text{Compute}_{\text{LIMIT}}$ during W_{HIGH}. This leads to congestion, a situation where records cannot be processed at their rate of their arrival. We repeated the experiment using each of the built-in ingestion policies (Table 2, Section 4.5).

Figure 18 shows the instantaneous throughput plotted on a timeline for each policy. Each figure also cites the *average ingestion latency* (L_{Avg}) during each workload-phase. It is desirable to maximize the *ingestion coverage* (Table 3(b)), minimize the average ingestion latency for each workload-phase and have $T_{\text{DONE}} \sim T_{\text{STOP}}$. The Basic and Spill policies were able to ingest all records (ingestion coverage = 1.0). However, T_{DONE} exceeded T_{STOP} due to the excess records created during W_{HIGH}. The Discard and Throttle policies had $T_{\text{DONE}} \sim T_{\text{STOP}}$, provided low ingestion latency but at the cost of reduced ingestion coverage (∼0.66). During W_{HIGH}, Throttle policy reduced the effective tweet arrival rate at each compute node from ∼600 tweets/second to ∼300 tweets/second (50% sampling rate)[2]. The Elastic policy acted differently by restructuring the pipeline to involve 20 compute operator instances during W_{HIGH} as the tweet arrival rate doubled. Each node then had two compute operator instances that provided a better utilization of the cores (4) on each node, effectively increasing the $\text{Compute}_{\text{LIMIT}}$ of the cluster. This helped provide a complete ingestion coverage (1.0), a minimum average ingestion latency for each workload-phase and had $T_{\text{DONE}} \sim T_{\text{STOP}}$. Recall that the MetaFeed operator dynamically evaluates the record arrival and processing rate at each core operator. Throttle and Elastic policies make use this periodic evaluation to derive the $\text{Compute}_{\text{LIMIT}}$ and adapt the sampling rate/degree of parallelism respectively.

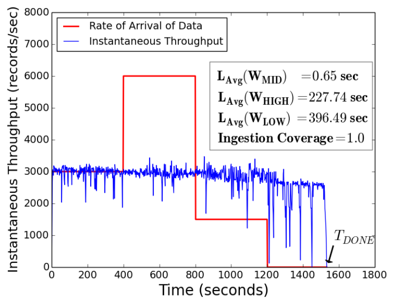
7.4 Comparison with Storm + MongoDB

An alternative way of supporting data ingestion today is to 'glue' together a streaming engine (e.g., Storm) with a persistent store (e.g., MongoDB) that supports queries over indexed semi-structured data. We used our 10 node cluster to host Storm and MongoDB. A Storm dataflow offers spouts (that act as sources of data) and bolts (that act as operators) that can be connected to form a dataflow. A spout implements a method, *nextTuple()* that is invoked by Storm in a 'pull-based' manner for obtaining the next record from a data

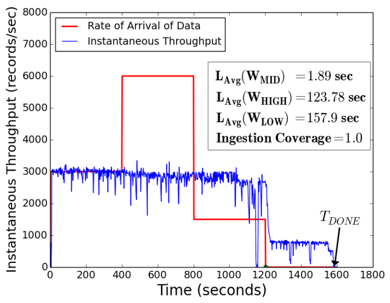
[2]Records arrive in fixed-size frames that contain varying number of records. Each frame is sampled to randomly select a subset of records for processing.



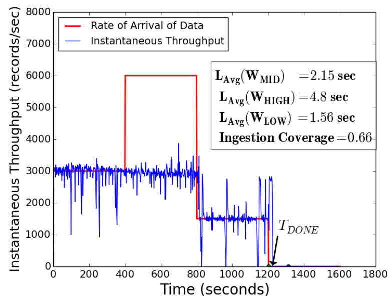
(a) Rate of Arrival of Data



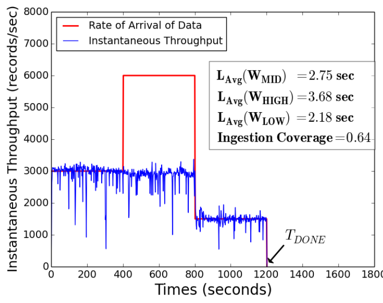
(b) Basic Policy



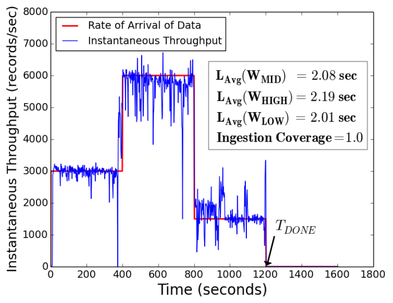
(c) Spill Policy



(d) Discard Policy



(e) Throttle Policy



(f) Elastic Policy

Figure 18: Impact of Ingestion Policy on Runtime Behavior

Table 3: Symbols and Metrics
(a) Symbol Definitions

Symbol	Definition
T_{start}, T_{stop}	Time when data source starts/stops pushing data
$T_{intake}(i)$	Time when Tweet(i) is received by the feed adaptor
$T_{indexed(i)}$	Time when Tweet(i) is indexed in storage
N_{total}	Total number of tweets received by feed adaptor
$N_{indexed}$	Total number of tweets indexed
T_{done}	Time when ingestion activity completes.

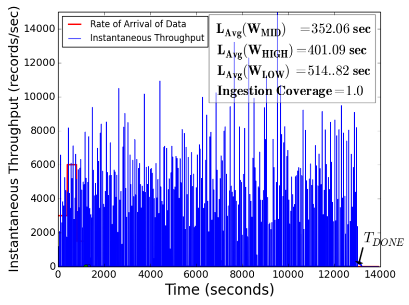
(b) Metric Definitions

Metric	Definition
Instantaneous Throughput (t)	$(N_{indexed}(t) - N_{indexed}(t-w))/w$, $w = 2$ *seconds*
Ingestion Latency (i)	$T_{indexed}(i) - T_{intake}(i)$
Ingestion Coverage	$N_{indexed}/N_{total}$

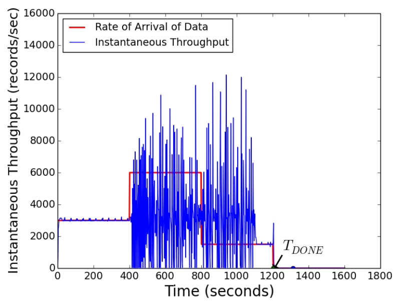
source. This method is not compatible with the common scenario of a 'push-based' ingestion where data continues to arrive from the data source at its natural rate. To support push-based ingestion, it is necessary to buffer the arriving records from the data source and then forward them to Storm on each invocation of the *nextTuple()* method. Another strategy commonly used by the community is to use yet another system — Redis, Thrift, or Kafka — as services (more 'gluing'!) so that records can be pushed to them and then a spout can pull them.

In contrast to our declarative support for defining/managing feeds, where the AsterixDB compiler constructs the dataflow, a Storm + MongoDB user must programmatically connect together spouts and bolts and statically specify the degree of parallelism for each. Storm does not offer elasticity, nor does it allow associating ingestion policies to customize the handling of congestion and failures. Interfacing with MongoDB requires the bolts to be parameterized with the locations of MongoDB Query Routers, which are processes running on specific nodes in a MongoDB cluster that accept insert statements/queries. The end user is thus required to understand the layout of the cluster and include specific information in the source code. Our 'glued' solution emulates the stages from an AsterixDB ingestion pipeline. The constructed dataflow involves a pair of spouts, each receiving records from a separate TweetGen instance. Each spout's output is randomly partitioned across a set of 10 bolts, one on each node. Each node also hosted a MongoDB Query Router to allow the co-located bolt to submit an insert statement to the local Query Router. Each node also hosted a MongoDB partition server. The MongoDB collection (dataset) was sharded (hashed by primary key) across the partition servers.

MongoDB provides a varying level of durability for writes. The lowest level (non-durable) allow submitting records for insertion asynchronously with no guarantees or notification of success. The Storm+MongoDB coupling then acts as a pure streaming engine with minimal overhead from (de)serialization of records. However, it becomes hard to reason about the consistency and durability offered by the system. The durable-write mode in MongoDB is a fair comparison with AsterixDB, as it provides ACID semantics for data ingestion. However, to provide a complete picture, we ran the workload of Figure 18(a) using both kinds of writes for MongoDB. The durable-write mode (Figure 19(a)) in the Storm+MongoDB coupling provides complete ingestion coverage. However, when compared to Basic, Spill and Elastic policies from AsterixDB (with similar ingestion coverage), the time taken for the ingestion activity to complete (T_{DONE} - T_{START}) increased by a factor of ten — meaning that Storm+MongoDB coupling was unable to keep up with the workload. The average ingestion latency observed in each workload-phase for Storm+MongoDB compared with the Elastic policy was worse by two orders of magnitude. To isolate the cause,



(a) Durable Write



(b) Non-Durable Write

Figure 19: Instantaneous Throughput for Storm+MongoDB.

we switched to using non-durable writes (Figure 19(b)) wherein the system behaves like a pure streaming engine with a de-coupled unreliable persistent store (asynchronous writes). We then obtained $T_{\text{DONE}} \sim T_{\text{STOP}}$. This ruled out inefficient streaming of records within Storm as a possible reason for the low throughput.

To better understand the results, we must consider the processing strategy used by MongoDB. MongoDB optimized for maximum single-record throughput and write-concurrency but at the cost of an increased wait time (~50ms) per write when full durability is requested. This created congestion at the output of the compute stage of our Storm+MongoDB combination and contributed to the high latency and low ingestion throughput. The situation is expected to worsen when 'at least once semantics' are required. Storm achieves such semantics by replaying a record if it does not traverse the dataflow within a specified time threshold. Owing to an increased wait time per write, additional failures would be assumed and records would begin to be replayed; this cycle can repeat endlessly, leading to system instability.

8. CONCLUSION

We have described the support for data feed management in AsterixDB (an open-source BDMS) and how it addresses the challenges involved in building a fault-tolerant data ingestion facility that scales through partitioned parallelism. We described how a feed may be defined and managed using a high-level language (AQL). A generic plug-and-play model helps AsterixDB cater to a wide variety of data sources and applications. We described the system's internal architecture and also provided a preliminary evaluation of the system, emphasizing its ability to scale to ingest increasingly large volumes of data and to handle failures during ingestion. A custom-built solution formed by 'gluing' together Storm and MongoDB was evaluated but did not compare well with the ingestion support provided by AsterixDB, neither in terms of user-experience nor its performance characteristics.

9. ACKNOWLEDGEMENTS

AsterixDB work has been supported by a UC Discovery grant and NSF IIS awards 0910989, 0910859, 0910820, and 0844574 and CNS awards 1305430, 1059436, and 1305253. Industrial supporters have included Amazon, eBay, Facebook, Google, HTC, Microsoft, Oracle Labs, and Yahoo!.

10. REFERENCES

[1] Asterixdb http://asterix.ics.uci.edu.

[2] "AsterixDB source," https://code.google.com/p/asterixdb.

[3] "Data on Big Data," http://marciaconner.com/blog/data-on-big-data/.

[4] "Informatica PowerCenter" http://www.informatica.com/in/etl/.

[5] "MongoDB," http://www.mongodb.org/.

[6] "Twitter's Storm," http://storm-project.net.

[7] D. Abadi et al. Aurora: A Data Stream Management System. *Proc. SIGMOD Conf.*, 2003.

[8] M. Balazinska et al. Fault-Tolerance in the Borealis Distributed Stream Processing System. In *Proc. SIGMOD Conf.*, 2005.

[9] A. Behm et al. ASTERIX: Towards a Scalable, Semi-structured Data Platform for Evolving-World Models. *Proc. DAPD*, 29, 2011.

[10] P. Bonnet et al. Towards Sensor Database Systems. *Mobile Data Management*, 2001.

[11] V. R. Borkar et al. Hyracks: A Flexible and Extensible Foundation for Data-Intensive Computing. *Proc. ICDE Conf.*, 2011.

[12] M. Carey et al. AsterixDB: A Scalable, Open Source BDMS. *Proc. VLDB Endow.*, 7(14), 2014.

[13] B. Gedik et al. SPADE: The System S Declarative Stream Processing Engine. *Proc. SIGMOD Conf.*, 2008.

[14] R. Grover and M. J. Carey. Scalable Fault-Tolerant Data Feeds in AsterixDB. *arXiv:1405.1705, CoRR*, 2014.

[15] L. Neumeyer et al. S4: Distributed Stream Computing Platform. *ICDM Workshops*, 2010.

[16] Z. Qian, Y. He, et al. Timestream: Reliable Stream Computation in the Cloud. *ACM EuroSys*, 2013.

[17] M. A. Shah et al. Highly Available, Fault-Tolerant, Parallel Dataflows. *Proc. SIGMOD Conf.*, 2004.

[18] M. Stonebraker. Operating System Support for Database Management. *Communication. ACM*, 24, 1981.

[19] Y. Xu et al. A Hadoop Based Distributed Loading Approach to Parallel Data Warehouses. *Proc. ICDE Conf.*, 2011.

The NPD Benchmark: Reality Check for OBDA Systems

Davide Lanti, Martin Rezk, Guohui Xiao, and Diego Calvanese
Faculty of Computer Science, Free University of Bozen-Bolzano
Piazza Domenicani 3, Bolzano, Italy
{dlanti,mrezk,xiao,calvanese}@inf.unibz.it

ABSTRACT

In the last decades we moved from a world in which an enterprise had one central database—rather small for todays' standards—to a world in which many different—and big—databases must interact and operate, providing the user an integrated and understandable view of the data. Ontology-Based Data Access (OBDA) is becoming a popular approach to cope with this new scenario. OBDA separates the user from the data sources by means of a conceptual view of the data (ontology) that provides clients with a convenient query vocabulary. The ontology is connected to the data sources through a declarative specification given in terms of mappings. Although prototype OBDA systems providing the ability to answer SPARQL queries over the ontology are available, a significant challenge remains when it comes to use these systems in industrial environments: performance. To properly evaluate OBDA systems, benchmarks tailored towards the requirements in this setting are needed. In this work, we propose a novel benchmark for OBDA systems based on real data coming from the oil industry: the Norwegian Petroleum Directorate (NPD) FactPages. Our benchmark comes with novel techniques to generate, from the NPD data, datasets of increasing size, taking into account the requirements dictated by the OBDA setting. We validate our benchmark on significant OBDA systems, showing that it is more adequate than previous benchmarks not tailored for OBDA.

1. INTRODUCTION

In the last decades we moved from a world in which an enterprise had one central database, to a world in which many different databases must interact and operate, providing the user an integrated view of the data. In this new setting five research areas in the database community became critical [1]: (i) scalable big/fast data infrastructures; (ii) ability to cope with diversity in the data management landscape; (iii) end-to-end processing and understanding of data; (iv) cloud services; and (v) managing the diverse roles of people in the data life cycle. Since the mid 2000s, *Ontology-Based Data Access* (OBDA) has become a popular approach used in three of these five areas— namely (ii), (iii), and (v).

In OBDA, queries are posed over a high-level conceptual view, and then translated into queries over a potentially very large (usually relational and federated) data source. The conceptual layer is given in the form of an ontology that defines a shared vocabulary, hides the structure of the data sources, and can enrich incomplete data with background knowledge. The ontology is connected to the data sources through a declarative specification given in terms of mappings that relate each (class and property) symbol in the ontology to a (SQL) view over (possibly federated) data. The W3C standard R2RML [8] was created with the goal of providing a standardized language for the specification of mappings in the OBDA setting. The ontology together with the mappings exposes a virtual instance (RDF graph) that can be queried using SPARQL, the standard query language in the Semantic Web community.

To make OBDA useful in an industrial setting, OBDA systems must provide answers in a reasonable amount of time, especially in the context of Big Data. However, most research in academia has focused on correct SPARQL-to-SQL translations, and expressivity of the ontology/mapping languages. Little effort (to the best of our knowledge) has been spent in systematically evaluating the performance of OBDA systems. To properly evaluate such performance, benchmarks tailored towards the requirements in this setting are needed. In particular, the benchmark should resemble a typical real-world industrial scenario in terms of the size of the data set, the complexity of the ontology, and the complexity of the queries. In this work, we propose a novel benchmark for OBDA systems based on the *Norwegian Petroleum Directorate* (NPD) *FactPages*[1]. The NPD FactPages contains information regarding the petroleum activities on the Norwegian continental shelf. Such information is actively used by oil companies, such as Statoil. The Factpages are synchronized with the NPD's databases on a daily basis. The NPD Ontology [26] has been mapped to the NPD FactPages and stored in a relational database[2]. The queries over such an ontology have been formulated by domain experts starting from an informal set of questions provided by regular users of the FactPages.

The contributions of this paper are as follows: *(1)* We identify requirements for benchmarking of OBDA systems in a real world scenario. *(2)* We identify requirements for data generation in the setting of OBDA. *(3)* We propose a benchmark that is compliant with the requirements identified. *(4)* We provide a data generator for OBDA together with an automatized testing platform. *(5)* We carry out an extensive evaluation using state-of-the-art OBDA systems and triple stores, revealing strength and weaknesses of OBDA.

This work extends the previous workshop publications [6, 17] with an automatized testing platform, with new experiments, and with a larger and more challenging query set, including also aggregate queries. This new query set highlights the importance of

[1]`http://factpages.npd.no/factpages/`
[2]`http://sws.ifi.uio.no/project/npd-v2/`

10.5441/002/edbt.2015.62

semantic query optimisation in the SPARQL-to-SQL translation phase.

The rest of the paper is structured as follows. In Section 2, we briefly survey other works related to benchmarking. In Section 3, we present the necessary requirements for an OBDA benchmark. In Section 4, we discuss the requirements for an OBDA instance generator. In Section 5, we present the NPD benchmark[3] and an associated relational database generator that gives rise to a virtual instance through the mapping; we call our generator *Virtual Instance Generator* (VIG). In Section 6, we describe a set of experiments performed using our benchmark over OBDA systems and triple stores. We conclude in Section 7.

2. RELATED WORK

Benchmarks are used to assess the quality of a system against a number of measures related to its design goals. Although OBDA systems have recently gained popularity, and the interest of a number of important enterprises like Siemens or Statoil (c.f. Optique Project[4]), no benchmark has yet been proposed in this setting. Although there are no guidelines nor benchmarks specific for OBDA, one must observe that these systems integrate both well-established database technologies and Semantic Web features. Driven by this observation, and given that both databases and knowledge-based systems have a vast literature on benchmarking, a natural starting point for deriving requirements for an OBDA benchmark is a synthesis of the requirements coming from both of these worlds.

For the databases world, two of the most popular benchmarks are the *Wisconsin Benchmark* [9] and the *TPC Benchmark*[5] The Wisconsin Benchmark specifies a single relation, and columns with different duplicates ratios allow one to easily manipulate the selectivity of the test queries. The TPC Benchmark comes in different flavors so as to test database systems in several popular scenarios, like transactions in an order-entry environment (TPC-C), or a brokerage firm with related customers (TPC-E). These benchmarks gained popularity for a number of reasons, prominently because they capture concrete use-cases coming from industry, they are simple to understand and run, and they provide metrics that allow one to clearly identify winners and losers (e.g., cost per transaction or query mixes per hour).

For the Semantic Web world, the situation is much less standardized, and a high number of benchmarks have been proposed. The most popular ones are *LUBM* [12], and *BSBM* [3], which are rather simple in the sense that they come either with a simple ontology, or with no ontology at all. These benchmarks do not allow one to properly test the performance of the reasoners in the context of complex and expressive ontologies—which are the vast majority when it comes to real-world applications. This aspect was pointed out in [31], where the authors proposed an extension of the LUBM benchmark (called *UOBM*) in order to overcome these limitations. Rather than proposing a new benchmark, [13] identifies a number of requirements for benchmarking knowledge base systems. In this work we follow a similar scheme, as we first identify a number of key requirements for OBDA benchmarking and then we validate our benchmark against those requirements.

A recent and relevant effort concerning benchmarks in the Semantic Web context comes from the *DBPedia Benchmark* [20]. In this benchmark, the authors propose a number of key features, like a data generator to produce "realistic" instances of increasing sizes, a number of real-world queries gathered from the DBPedia SPARQL endpoint, and the DBPedia ontology. Although this is an extremely valuable effort in the context of knowledge base systems, there are still a number of characteristics that make the DBPedia Benchmark unsuitable for OBDA benchmarking (see Section 3).

The last effort in order of time comes from the attempt to create a council like TPC in the context of graph-like data management technologies, like Graph Data Base Management Systems or systems based on RDF graphs. The council is called *LDBC*[6], and it has so far produced two benchmarks related to data publishing and social use-cases. This is a remarkable effort, however the ontologies used in these benchmarks are in RDFS, rather than full OWL 2 QL; therefore, they might miss to test important OBDA-specific pitfalls, such as reasoning w.r.t. existentials [23].

3. REQUIREMENTS FOR BENCHMARKING OBDA

In this section, we study the requirements that benchmark to evaluate OBDA systems has to satisfy. In order to define these requirements, we first recall that the three fundamental components of such systems are: *(i)* the *conceptual layer* constituted by the ontology; *(ii)* the *data layer* provided by the data sources; and *(iii)* the *mapping layer* containing the declarative specification relating each (class and property) symbol in the ontology to an (SQL) view over (possibly federated) data. It is this mapping layer that decouples the virtual instance being queried, from the physical data stored in the data sources. Observe that triple stores cannot be considered as full-fledged OBDA systems, since they do not make a distinction between physical and virtual layer. However, given that both OBDA systems and triple stores are considered as (usually SPARQL) query answering systems, we consider it important that a benchmark for OBDA can also be used to evaluate triple stores. Also, since one of the components of an OBDA system is an ontology, the requirements we identify include those to evaluate general knowledge based systems [19, 13, 30]. However, due to the additional components, there are also notable differences.

Typically OBDA systems follow the workflow below for query answering:

1. *Starting phase*. The system loads the ontology and the mappings, and performs some auxiliary tasks needed to process/answer queries in a later stage. Depending on the system, this phase might be critical, since it might include some reasoning tasks, for example *inference materialization* or the embedding of the inferences into the mappings (*T-mappings* [22]).

2. *Query rewriting phase*. The input query is *rewritten* to a (typically more complex) query that takes into account the inferences induced by the intensional level of the ontology (we forward the interested reader to [4, 15]).

3. *Query translation* (or *unfolding*) *phase*. The rewritten query is translated into a query over the data sources. This is the phase where the mapping layer comes into play [21].

4. *Query execution phase*. The data query is executed over the original data source, answers are produced according to the data source schema, and are translated into answers in terms of the ontology vocabulary and RDF data types, thus obtaining an answer to the original input query.

Note that a variation of the above workflow has actually been proposed in [19], but without identifying a distinct starting phase, and

[3]`https://github.com/ontop/npd-benchmark/`
[4]`http://www.optique-project.eu/`
[5]`http://www.tpc.org/`
[6]`http://www.ldbcouncil.org/`

Table 1: Measures for OBDA

name	triple store	related to phase
Performance Metrics		
Loading Time	(**T**)	1
Rewriting Time	($\mathbf{T}^*$)	2
Unfolding Time	—	3
Query execution time	(**T**)	4
Overall response time	(**T**)	2, 3, 4
Quality Metrics		
Simplicity R Query	($\mathbf{T}^*$)	2
Simplicity U Query	—	3
Weight of R+U	($\mathbf{T}^*$)	2, 3, 4

instead singling out from query execution a result translation phase. It is critical to notice that although optimisation is not mentioned in this workflow, it is the most challenging part in the query answering process, and *definitely essential* to make OBDA applicable in production environments.

There are several approaches to deal with Phase 2 [15, 29]. The most challenging task in this phase is to deal with existentials in the right-hand side of ontology axioms. These axioms infer unnamed individuals in the virtual instance that cannot be retrieved as part of the answer, but can affect the evaluation of the query. An approach that has proved to produce good results in practice is the *tree-witness rewriting* technique, for which we refer to [15]. For us, it is only important to observe that *tree-witnesses* lead to an extension of the original query to account for matching in the existentially implied part of the virtual instance. Below, we take the number of tree-witnesses identified in Phase 2 as one of the parameters to measure the complexity of the combination ontology/query. Since existentials do not occur very often in practice [15], and can produce an exponential blow-up in the query size, some systems allow one to turn off the part of Phase 2 that deals with *reasoning with respect to existentials*.

Ideally, an OBDA benchmark should provide *meaningful* measures for each of these phases. Unfortunately, such a fine-grained analysis is not always possible, for instance because the system comes as a black-box with proprietary code with no APIs providing the necessary information, e.g., the access to the rewritten query; or because the system combines more phases into one, e.g., query rewriting and query translation. Based on the above phases, we identify in Table 1 the measures important for *evaluating* OBDA systems. The meaning of the *Performance Metrics* should be clear from their names; instead, we will give a brief explanation of the meaning of the *Quality Metrics*:

- *Simplicity R Query*. Simplicity of the rewritten query in terms of language dependent measures, like the *number of rules* in case the rewritten query is a Datalog program. In addition, one can include system-dependent features, e.g., the number of tree-witnesses in *Ontop*.
- *Simplicity U Query*. This measures the simplicity of the query over the data source, including relevant SQL-specific metrics like the number of joins/left-join, the number of inner queries, etc.
- *Weight of R+U*. It is the cost of the construction of the SQL query divided by the overall cost.

We label with **(T)** those measures that are also valid for triple stores, and with ($\mathbf{T}^*$) those that are valid only if the triple store is based on query rewriting (e.g., Stardog). Notice that the two *Simplicity* measures, even when retrievable, are not always suitable for *comparing* different OBDA systems. For example, it might not be possible to compare the simplicity of queries in the various phases, e.g., when such queries are expressed in different languages.

With these measures in mind, the different components of the benchmark should be designed so as to reveal strengths and weaknesses of a system in each phase. The conclusions drawn from the benchmark are more significant if the benchmark resembles a typical real-world scenario in terms of the complexity of the ontology and queries, and the size of the data set. Therefore, we consider the benchmark requirements in Table 2.

The current benchmarks available for OBDA do not meet several of the requirements above. Next we list some of the best known benchmarks and their shortcomings when it comes to evaluating OBDA systems. We show general statistics in Table 3.

Adolena: Designed in order to extend the South African National Accessibility Portal [14] with OBDA capabilities. It provides a rich class hierarchy, but a quite poor structure for properties. This means that queries over this ontology will usually be devoid of tree-witnesses. No data-generator is included, nor mappings.
Requirements Missing: O1, Q2, D2, S1

LUBM: The Lehigh University Benchmark (LUBM) [12] consists of a university domain ontology, data, and queries. For data generation, the UBA (Univ-Bench Artificial) data generator is available. However, the ontology is rather small, and the benchmark is not tailored towards OBDA, since no mappings to a (relational) data source are provided.
Requirements Missing: O1, Q2, M1, M2, D1

DBpedia: The DBpedia Benchmark consists of a relatively large—yet, simple[7]—ontology, a set of user queries chosen among the most popular queries posed against the DBpedia[8] SPARQL endpoint, and a synthetic RDF data generator able to generate data having properties similar to the real-world data. This benchmark is specifically tailored to triple stores, and as such it does not provide any OBDA specific components like R2RML mappings, or a data set in the form of a relational database.
Requirements Missing: O1, O2, Q2, M1, M2

BSBM: The Berlin SPARQL Benchmark [3] is built around an e-commerce use case. It has a data generator that allows one to configure the data size (in triples), but there is no ontology to measure reasoning tasks, and the queries are rather simple. Moreover, the data is fully artificial.
Requirements Missing: O1, O2, Q2, M1, M2, D1,

FishMark: FishMark [2] collects comprehensive information about finned fish species. This benchmark is based on the FishBase real world dataset, and the queries are extracted from popular user SQL queries over FishBase; they are more complex than those from BSBM. However, the benchmark comes neither with mappings nor with a data generator. The data size is rather small (≈20M triples).
Requirements Missing: O1, D2, S1

A specific challenge comes from requirements **D1** and **D2**, i.e., given an initial real-world dataset, together with a rich ontology and mappings, expand the dataset in such a way that it populates the

[7]In particular, it is not suitable for reasoning w.r.t. existentials.
[8]`http://dbpedia.org/sparql/`

Table 2: Benchmark Requirements

O1 The **ontology** should include rich hierarchies of classes and properties.	**Q1** The **query set** should be based on actual user queries.	**M1** The **mappings** should be defined for elements of most hierarchies.
O2 The **ontology** should contain a rich set of axioms that infer new objects and could lead to inconsistency, in order to test the reasoner capabilities.	**Q2** The **query set** should be complex enough to challenge the query rewriter.	**M2** The **mappings** should contain redundancies, and suboptimal SQL queries to test optimizations.
D1 The **virtual instance** should be based on real world data.	**D2** The size of the **virtual instance** should be tunable.	**S1** The **languages** of the ontology, mapping, and query should be *standard*, i.e., based on R2RML, SPARQL, and OWL respectively.

Table 3: Popular Benchmark Ontologies: Statistics

	Ontology Stats. (Total)			Queries Stats. (Max)		
name	#classes	#obj/data_prop	#i-axioms	#joins	#opt	#tw
adolena	141	16	189	5	0	0
lubm	43	32	91	7	0	0
dbpedia	530	2148	3836	7	8	0
bsbm	8	40	0	14	4	0
fishmark	11	94	174	24	12	0

virtual instance in a sensible way (i.e., coherently with the ontology constraints and relevant statistical properties of the initial dataset). We address this problem in the next section.

4. REQUIREMENTS FOR DATA GENERATION

In this section, we present the requirements for an OBDA data generator, under the assumption that we have an initial database that can be used as a seed to understand the distribution of the data that needs to be increased. To ease the presentation, we illustrate the main issues that arise in this context with an example.

EXAMPLE 4.1. Consider a database $\mathcal{D}$ made of four tables, namely `TEmployee`, `TAssignment`, `TSellsProduct`, and `TProduct`. Table 4 shows a fragment of the content of the tables and their schemas, where **bold font** denotes primary keys and the foreign keys are *in italics*. We assume that every employee sells the majority of the products, hence the table `TSellsProduct` contains roughly the cross product of the tables `TEmployee` and `TProduct`. Next we present only a fragment of the data.

Table 4: Database $\mathcal{D}$

`TEmployee`

id	name	branch
1	John	B1
2	Lisa	B1

`TAssignment`

branch	task
B1	task1
B1	task2
B2	task1
B2	task2

`TSellsProduct`

id	*product*
1	p1
2	p2
1	p2
2	p3

`TProduct`

product	size
p1	big
p2	big
p3	small
p4	big

Table 5 defines the set $\mathcal{M}$ of mapping assertions used to populate the ontology concepts `:Employee`, `:Branch`, and `:ProductSize`, plus the object properties `:SellsProduct` and `:AssignedTo`.

The virtual instance corresponding to the database $\mathcal{D}$ and mappings $\mathcal{M}$ includes the following RDF triples:

```
:1  rdf:type   :Employee.
:2  rdf:type   :Employee.
:1  :SellsProduct  :p1.
:1  :SellsProduct  :p2.
:2  :AssignedTo    :t1.
```

Suppose now we want to increase the *virtual* RDF graph by a *growth-factor* of 2. Observe that this is not as simple as doubling the number of triples in every concept and property, or the number of tuples in every database relation. Let us first analyze the behavior of some of the ontology elements w.r.t. this aspect, and then how the mappings to the database come into play.

- `:ProductSize`: This concept will contain two individ-

Table 5: Mappings $\mathcal{M}$

$\mathcal{M}_1$	`:{id} rdf:type :Employee`	$\leftarrow$	`SELECT id from TEmployee`
$\mathcal{M}_2$	`:{branch} rdf:type :Branch`	$\leftarrow$	`SELECT branch FROM TAssignments`
$\mathcal{M}_3$	`:{branch} rdf:type :Branch`	$\leftarrow$	`SELECT branch FROM TEmployee`
$\mathcal{M}_4$	`:{id} :SellsProduct :{product}`	$\leftarrow$	`SELECT id, product FROM TSellsProduct`
$\mathcal{M}_5$	`:{size} rdf:type :ProductSize`	$\leftarrow$	`SELECT size FROM TProduct`
$\mathcal{M}_6$	`:{id} :AssignedTo :{task}`	$\leftarrow$	`SELECT id, task FROM TEmployee NATURAL JOIN TAssignments`

uals, namely `:small` and `:big`, independently of the growth-factor. Therefore, the virtual instances of the concept should not be increased when the RDF graph is extended.

- `:Employee` and `:Branch`: Since these classes do not depend on other properties, and since they are not intrinsically constant, we expect their size to grow linearly with the growth-factor.
- `:AssignedTo`: Since this property represents a cartesian product, we expect its size to grow roughly quadratically with the growth-factor.
- `:SellsProduct`: The size of this property grows with the product of the numbers of `:Employees` and `:Products`. Therefore, when we double these numbers, the size of `:SellsProduct` will roughly quadruplicate.

In fact, the above considerations show that we do not have *one* uniform growth-factor for the ontology elements. Our choice is to characterize the growth in terms of the increase in size of those concepts in the ontology that are not intrinsically constant (e.g., `:ProductSize`), and that do not "depend" on any other concept, considering the semantics of the domain of interest (e.g., `:Employee`). We take this as measure for the growth-factor.

The problem of understanding how to generate from a given RDF graph new additional triples coherently with the domain semantics is addressed in [30, 20]. The algorithm in [30] starts from an initial RDF graph and produces a new RDF graph, considering key features of the original graph (e.g., the distribution of connections among individuals). However, this approach, and all approaches producing RDF graphs in general, cannot be directly applied to the context of OBDA, where the RDF graph is virtual and generated from a relational database. Trying to apply these approaches indirectly, by first producing a "realistic" virtual RDF graph and then trying to reflect the virtual data into the physical (relational) datasource, is far from trivial due to the correlations in the underlying data. This problem, indeed, is closely related to the *view update problem* [7], where each class (resp., role or data property) can be seen as a *view* on the underlying physical data. The view update problem is known to be challenging and actually decidable only for a very restricted class of queries used in the mappings [10]. Note, however, that our setting does not necessarily require to fully solve the view update problem, since we are interested in obtaining a physical instance that gives rise to a virtual instance with certain statistics, but not necessarily to a specific given virtual instance. The problem we are facing nevertheless remains challenging, and requires further research. We illustrate the difficulties that one encounters again on our example.

- The property `:SellsProduct` grows linearly w.r.t. the size of the table `TSellsProduct`, hence also this table has to grow quadratically with the growth-factor. Since `TSellsProduct` has foreign keys from the tables `TEmployee` and `TProduct`, to preserve the fact that every employee must be connected to every product, the two tables `TEmployee` and `TProduct` have both to grow linearly. It is worth noting that, to produce one `:SellsProduct` triple in the virtual instance, we have to insert three tuples in the database.
- Since the `:Branch` concept should grow linearly with the growth-factor, in order to preserve the duplicates ratio in the `TAssignment.branch` column then also the `TAssignment` table should grow linearly, and there should always be less branches than employees in `TEmployee`.
- Since `:ProductSize` does not grow, the attribute `Size` must contain only two values, despite the linear growth of `TProduct`. ■

The previous example illustrated several challenges that need to be addressed by the generator regarding the *analysis* of the virtual and physical data, and the *insertion* of values in the database. Our goal is to generate a synthetic virtual graph where the cost of the queries is as similar as possible to the cost that the same query would have in a real-world virtual graph of comparable size. Observe that the same virtual graph can correspond to different database instances, that could behave very differently w.r.t. the cost of SQL query evaluation. Therefore, in order to keep the cost of the SPARQL query "realistic", we need to keep the cost of the translated SQL "realistic" as well.

We are interested in data generators that perform an analysis phase on real-world data, and that use the statistical information learned in the analysis phase for their task. We present first in Table 6 the measures that are relevant in the analysis phase. We then derive the requirements for the data generator by organizing them in two categories: one for the analysis phase, and one for the generation phase.

Measures for the Analysis Phase.

The measures are summarized in Table 6. The table is divided in three parts:

The top part refers to measures relevant at virtual instance level, i.e., those capturing the shape of the virtual instance. *Virtual Multiplicity Distribution* **(VMD)** describes the multiplicity of the properties, i.e., given a property p, and a number k, the VMD is the probability that a node n in the domain of p is connected to k elements through p. For instance, the VMD of `:AssignedTo` assigns probability 1 to the number 2. Observe that VMD is affected by the growth of the database (e.g., if the growth factor is 2, and the number of "tasks" grows linearly then the VMD of `:AssignedTo` assigns probability 1 to the number 4). *Virtual Growth* **(VG)** is the expected growth for each ontology term w.r.t. the growth-factor. For instance, the virtual growth of `:AssignedTo` is quadratic.

The middle part refers to measures at the physical level that affect the VMD of the properties through the mappings. They are based on the sets of attributes of a table used in the mappings to

Table 6: Relevant measures at the virtual and physical instance level

Measures affecting the virtual instance level	
Virtual Multiplicity Distribution (VMD)	**Virtual Growth (VG)**
Multiplicity distribution of the properties in the virtual graph.	Function describing how fast concepts (resp., role/data properties) grow w.r.t. the growth-factor.
Measures affecting virtual multiplicity distribution	
Intra-table IGA Multiplicity Distribution (Intra-MD)	**Inter-table IGA Multiplicity Distribution (Inter-MD)**
Multiplicity distribution between IGAs belonging to the same table and generating objects connected through a virtual property.	Multiplicity distribution between IGAs belonging to different tables and connected through a virtual property.
Measures affecting RDBMS performance and virtual growth	
IGA Duplication (D)	
Repeated IGAs	
Intra-table IGA-pair Duplication (Intra-D)	**Inter-table IGA-pair Duplication (Inter-D)**
Repeated pairs of intra-table correlated IGAs.	Repeated pairs of inter-table correlated IGAs.

define individuals and values in the ontology. We call such a set of attributes an *IGA* (individual-generating attributes). We say that two IGAs are *related* if and only if they occur in the same mapping defining the subject and the object of a property. Establishing the relevant statistics requires to identify pairs of IGAs through mapping analysis. *Intra-table Multiplicity Distribution* **(Intra-MD)** is defined for two related IGAs of the same table, both mapped to individuals/values at the virtual level. It is defined for tuples over the IGAs in the same way as the VMD is defined for individuals. For instance, the Intra-MD for the IGAs {id} and {product} with respect to the property `:SellsProduct` assigns probability 1 to the number 2. *Inter-table Multiplicity Distribution* **(Inter-MD)** is defined for related IGAs belonging to two different tables. It is calculated like the Intra-MD but over the joins specified in the mappings, e.g., the join of `TEmployee` and `TAssignment`.

The bottom part refers to measures at the physical level that do not affect VMD, but that influence growth at the virtual level and the overall performance of the system. Specifically, *IGA Duplication* **(D)** measures the ratio of identical copies of tuples over an IGA not occurring in a property definition, while (Intra-table and Inter-table) *IGA-pair Duplication* **(Intra-D** and **Inter-D)** are measured as the ratio of identical copies of a tuple over two related IGAs. For instance, the IGA Duplication for the IGA `TAssignment.branch` is $1/2$ (half of the tuples are duplicated).

Now we are ready to list the requirements for a data generator for OBDA systems.

Requirements for the Analysis Phase.

The generator should be able to analyze the physical instance and the mappings, in order to acquire statistics to assess the measures identified in Table 6.

Requirements for the Generation Phase.

We list now important requirements for the generation of physical data that gives rise through the mappings to the desired virtual data instance.

Tunable. The user must be able to specify a growth factor according to which the virtual instance should be populated.

Virtually Sound. The virtual instance corresponding to the generated physical data must meet the statistics discovered during the analysis phase and that are relevant at the virtual instance level.

Physically Sound. The generated physical instance must meet the statistics discovered during the analysis phase and that are relevant at the physical instance level.

Database Compliant. The generator must generate data that does not violate the constraints of the RDBMS engine—e.g., primary keys, foreign keys, constraints on datatypes, etc.

Fast. The generator must be able to produce a vast amount of data in a reasonable amount of time (e.g., 1 day for generating an amount of data sufficient to push the limits of the considered RDBMS system). This requirement is important because OBDA systems are expected to operate in the context of "Big-Data" [11].

5. NPD BENCHMARK

The Norwegian Petroleum Directorate[9] (NPD) is a governmental organisation whose main objective is to contribute to maximize the value that society can obtain from the oil and gas activities. The initial dataset that we use is the *NPD FactPages* (see Footnote 1), containing information regarding the petroleum activities on the Norwegian Continental Shelf (NCS).

The NPD benchmark consists of an an initial dataset reflecting the content of the FactPages, an ontology, a query set, a set of mappings, a data generator able to meaningfully increase the size of the initial dataset, and an automated testing platform.[10] The ontology, the query set, and the mappings to the dataset have all been developed at the University of Oslo [26], and are freely available online (see Footnote 2). We adapted each of these, fixing some minor inconsistencies, adding missing mappings, and slightly modifying the query set to make the queries more suitable for an OBDA benchmark. Next we provide more details on each of these items.

The Dataset.

The data from FactPages has been translated from CSV files into a structured database [26]. The obtained schema consists of 70 tables with 276 distinct columns ($\approx$1000 columns in total), and 94 foreign keys. The schemas of the tables overlap in the sense that several attributes are replicated in several tables. In fact, there are tables with more than 100 columns. The total size of the initial dataset is $\approx$50Mb.

The Ontology.

The ontology contains OWL axioms specifying comprehensive information about the underlying concepts in the dataset; in par-

[9] `http://www.npd.no/en/`

[10] `https://github.com/ontop/npd-benchmark/`

Table 7: Statistics for the queries considered in the benchmark

query	#join	#tw	max(#subcls)	# opts	Agg	Filt.	Mod.
Q1	4	0	0	0	N	Y	N
Q2	5	0	0	0	N	Y	N
Q3	3	0	0	0	N	Y	Y
Q4	5	0	0	0	N	Y	Y
Q5	5	0	0	0	N	Y	Y
Q6	6	2	23	0	N	Y	Y
Q7	7	0	0	0	N	Y	N
Q8	3	0	0	0	N	Y	N
Q9	3	0	38	0	N	Y	Y
Q10	2	0	0	0	N	Y	Y
Q11	7	2	23	0	N	Y	Y
Q12	8	4	23	0	N	Y	Y
Q13	2	0	0	2	N	Y	N
Q14	2	0	0	2	N	Y	N
Q15	4	-	0	0	Y	Y	N
Q16	3	-	0	0	Y	Y	N
Q17	8	-	0	0	Y	N	Y
Q18	4	-	0	0	Y	N	N
Q19	8	-	0	0	Y	N	N
Q20	3	-	0	0	Y	N	N
Q21	3	-	0	0	Y	N	N

ticular, the NPD ontology presents rich hierarchies of classes and properties, axioms that infer new objects, and disjointness assertions. We took the OWL 2 QL fragment of this ontology, and we obtained 343 classes, 142 object properties, 238 data properties, 1451 axioms, and a maximum hierarchy depth of 10. Since we are interested in benchmarking OBDA systems that are able to rewrite queries over the ontology into SQL-queries that can be evaluated by a relational DBMS, we concentrate here on the OWL 2 QL profile[11] of OWL, which guarantees rewritability of unions of conjunctive queries (see, e.g., [4]). This ontology is suitable for benchmarking reasoning tasks, given that *(i)* it is a representative [18] and complex real-world ontology in terms of number of classes and maximum depth of the class hierarchy (hence, it allows for reasoning w.r.t. class hierarchies); *(ii)* it is complex w.r.t. properties, therefore it allows for reasoning w.r.t. existentials.
From the previous facts, it follows that the ontology satisfies requirements **O1**, **O2**, **S1**.

The Query Set.

The original NPD SPARQL query set contains 20 queries obtained by interviewing users of the NPD dataset. Starting from the original NPD query set, we devised 21 queries having different degrees of complexity (see Table 7). We also fixed some minor issues in the queries/ontology, e.g., the absence in the ontology of certain concepts present in the queries, fixing type inconsistencies, and flattening of nested sub-queries. In particular, observe that most complex queries involve both classes with a rich hierarchy and tree witnesses, which means that they are particularly suitable for testing the reasoner capabilities. Aggregates are also a source of complexity in the context of OBDA, since they increase the complexity of the semantic query optimisation tasks. These aggregate queries were not part of the first draft of this benchmark [6, 17], and they either add aggregates to queries without them—for instance, q15 is obtained from q1—or they are a fragment of aggregate queries in the original NPD query set—for instance, q17 and q19. Next, we provide some example queries from the benchmark.

The following query (q6) is a query with tree-witnesses that asks for the wellbores, their length, and the companies that completed the drilling of the wellbore after 2008, and sampled more than 50m of cores.

```
SELECT DISTINCT ?wellbore (?length AS ?lenghtM)
                ?company ?year
WHERE {
  ?wc npdv:coreForWellbore
  [ rdf:type npdv:Wellbore ;
    npdv:name ?wellbore ;
    npdv:wellboreCompletionYear    ?year ;
    npdv:drillingOperatorCompany
    [npdv:name ?company ]] .
  { ?wc npdv:coresTotalLength ?length }
  FILTER(?year >= "2008"^^xsd:integer &&
        ?length > 50
  )}
```

When existential reasoning is enabled, query q6 produces 2 tree-witnesses, and gets rewritten into a union of 73 intermediate queries (query rewriting phase). The tree-witnesses arise due to existential axioms containing `npdv:Wellbore` and `npdv:coreForWellbore`.

The following query (q16) is a simple aggregate query that asks for the number of production licenses granted after year 2000.

```
SELECT (COUNT(?licence ) AS ?licnumber)
WHERE { [ ] a npdv:ProductionLicence ;
        npdv:name ?licence ;
        npdv:dateLicenceGranted ?dateGranted ;
        FILTER(?dateGranted > 2000) }
```

From the previous facts, it follows that the queries satisfy requirements **Q1**, **Q2**, **S1**.

The Mappings.

The R2RML mapping consists of 1190 assertions mapping a total of 464 among classes, objects properties, and data properties. The SQL queries in the mappings count an average of 2.6 unions of select-project-join queries (SPJ), with 1.7 joins per SPJ. We observe that the mappings have not been optimized to take full advantage of an OBDA framework, e.g., by trying to minimize the number of mappings that refer to the same ontology class or property, so as to reduce the size of the SQL query generated by unfolding the mapping. This gives the opportunity to the OBDA system to apply different optimization on the mappings at loading time.
From the previous facts, it follows that the mappings satisfies requirements **M1**, **M2**, **S1**.

Automatized Testing Platform.

The benchmark comes with a testing platform (called OBDA Mixer[12]) that allows one to automatize the runs of the tests and the collection of results. Mixer comes in the form of an easily extensible Java project, which can be extended to work with other OBDA systems as long as they provide a Java API and public interfaces able to return interesting statistics (e.g., unfolding or rewriting times).

5.1 VIG: The Data Generator

Next we present the Virtual Instances Generator (VIG) that we implemented in the NPD Benchmark. VIG produces a virtual instance by inserting data into the original database. The generator is general in the sense that, although it currently works with the NPD database, it can produce data also starting from instances different from NPD. The algorithm can be divided into two main phases, namely *(i)* an *analysis phase*, where statistics for relevant measures on the real-world data are identified, and *(ii)* a *generation phase*, where data is produced according to the identified statistics.

[11] `http://www.w3.org/TR/owl2-profiles/`

[12] `https://github.com/ontop/obda-mixer`

VIG starts from a non-empty database D. Given a growth factor g, VIG generates a new database D' such that $|T'| \approx |T| \cdot (1+g)$, for each table T of D (where $|T|$ denotes the number of tuples of T). The size is approximated since, due to foreign key constraints, some tables might require the addition of extra tuples. In other words, the current implementation of the data generator assumes that the size of each table T grows accordingly to the growth factor. This rules out for example the case when the size of a table T depends on the cartesian product of two foreign keys (as in Example 4.1), since in this case the size of T depends quadratically on the sizes of the referred tables. In the case of NPD, however, there are no such tables and therefore the growth for each table is at most linear. Observe that the chosen generation strategy does not imply that every concept or property at the virtual level grows as the growth factor, since the growth depends not only on the content of the tables but also on the shape of the SQL queries defined in the mappings (c.f. Example 4.1).

We now describe how VIG approximates the measures described in Table 6.

Measures (D), (Intra-D).

We compute (an approximation for) these measures by *Duplicate Values Discovery.* For each column $T.C$ of a table $T \in D$, VIG discovers the *duplicate ratio* for values contained in that column. The *duplicate ratio* is the ratio $(\|T.C\| - |T.C|)/\|T.C\|$, where $\|T.C\|$ denotes the number of values in the column $T.C$, and $|T.C|$ denotes the number of distinct values in $T.C$. A duplicate ratio "close to 1" indicates that the content of the column is essentially *independent* from the size of the database, and it should not be increased by the data generator.

Measures (Intra-MD), (Inter-MD), (Inter-D).

Instead of computing (an approximation for) these measures, VIG identifies the domain of each attribute. That is, for each column $T.C$ in a table T, VIG analyzes the content of $T.C$ in order to decide the range of values from which *fresh* non-duplicate values can be chosen. More specifically, if the domain of $T.C$ is `String` or simply unordered (e.g., polygons), then a random fresh value is generated. Instead, if the domain is a total order, then fresh values can be chosen from the non-duplicate values in the interval $[\min(T.C), \max(T.C)]$ or in the range of values adjacent to it. Observe that this helps in maintaining the domain of a column similar to the original one, and this in turn helps in maintaining Intra- and Inter-table Multiplicity Distribution. VIG also preserves standard database constraints, like primary keys, foreign keys, and datatypes, that during the generation phase will help in preserving the IGA Multiplicity Distribution. For instance, VIG analyses the loops in foreign key dependencies in the database. Let $T_1 \to T_2$ denote the presence of a foreign key from table T_1 to table T_2. In case of a cycle $T_1 \to T_2 \to \cdots \to T_k \to T_1$, inserting a tuple in T_1 could potentially trigger an infinite number of insertions. VIG performs an analysis on the values contained in the columns involved by the dependencies and discovers the maximum number of insertions that can be performed in the generation phase.
Next we describe the generation phase, and how it meets some of the requirements given in Section 6.

Duplicate Values Generation.

VIG inserts duplicates in each column according to the duplicate ratio discovered in the analysis phase. Each duplicate is chosen with a uniform probability distribution. This ensures, for those concepts that are not dependent from other concepts and whose individual are "constructed" from a single database column, a growth that is equal to the growth factor. In addition, it prevents intrinsically constant concepts from being increased (by never picking a fresh value in those columns where the duplicates ratio is close to 1). Finally, it helps keeping the sizes for join result sets "realistic" [28]. This is true in particular for the NPD database, where almost every join is realized by a single equality on two columns.
Requirement: Physically/Virtually Sound.

Fresh Values Generation.

For each column, VIG picks fresh non-duplicate values from the interval $\mathcal{I}$ discovered during the analysis phase. If the number of values to insert exceeds the number of different fresh values that can be chosen from the interval $\mathcal{I}$, then values outside $\mathcal{I}$ are allowed. The choices for the generation of new values guarantees that columns always contain values "close" to those already present in the column. This ensures that the number of individuals for concepts based on comparisons grows accordingly to the growth factor.
Requirement: Physically/Virtually Sound.

Metadata Constraints.

VIG generates values that do not violate the constraints of the underlying database, like primary keys, foreign keys, or type constraints. The NPD database makes use of geometric datatypes available in MySQL. Some of them come with constraints, e.g., a polygon is a closed non-intersecting line composed of a finite number of straight lines. For each geometric column in the database, VIG first identifies the minimal rectangular region of space enclosing all the values in the column, and then it generates values in that region. This ensures that artificially generated geometric values will fall in the result sets of selection queries.
Requirement: Database Compliant/Virtually Sound.

Length of Chase Cycles.

In case a cycle of foreign key dependencies was identified during the analysis phase, then VIG stops the chain of insertions according to the boundaries identified in the analysis phase, while ensuring that no foreign key constraint is violated. This is done by inserting either a duplicate or a null in those columns that have a foreign key dependency.
Requirement: Database Compliant.

Furthermore, VIG allows the user to tune the growth factor, and the generation process is considerably fast, for instance, it takes ≈10hrs to generate 130 Gb of data.

5.2 Validation of the Data Generator

In this section we perform a qualitative analysis of the virtual instances obtained using VIG. We focus our analysis on those concepts and properties that either are supposed to grow linearly w.r.t. the growth factor or are supposed not to grow at all. These are 138 concepts, 28 object properties, and 226 data properties.

We report in Table 8 the growth of the ontology elements w.r.t. the growth of databases produced by VIG and by a purely random generator. The first column indicates the type of ontology elements being analyzed, and the growth factor g (e.g., "class_npd2" refers to the population of classes for the database incremented with a growth factor $g = 2$). The columns under "avg dev" show the average deviation of the actual growth from the expected growth, in terms of percentage of the expected growth. The remaining columns report the number and percentage of concepts (resp., object/data properties) for which the deviation was greater than 50%.

Concerning concepts, VIG behaves close to optimally. For properties, the difference between the expected virtual growth and the

Table 8: Comparison between VIG and a random data generator

	avg dev		err ≥50% (absolute)		err ≥50% (relative)	
type_db	heuristic	random	heuristic	random	heuristic	random
class_npd2	3.24%	370.08%	2	67	1.45%	48.55%
class_npd10	6.19%	505.02%	3	67	2.17%	48.55%
obj_npd2	87.48%	648.22%	8	12	28.57%	42.86%
obj_npd10	90.19%	883.92%	8	12	28.57%	42.86%
data_npd2	39.38%	96.30%	20	46	8.85%	20.35%
data_npd10	53.49%	131.17%	28	50	12.39%	22.12%

Table 9: Tractable queries (MySQL)

db	avg(ex_time) msec.	avg(out_time) msec.	avg(res_size) msec.	qmpH	#(triples)
NPD	44	102	15960	2167.37	≈2M
NPD2	70	182	30701	1528.01	≈6M
NPD10	148	463	81770	803.86	≈25M
NPD50	338	1001	186047	346.87	≈116M
NPD100	547	1361	249902	217.36	≈220M
NPD500	2415	5746	943676	57.80	≈1.4B
NPD1500	6740	18582	2575679	17.66	≈4B

Table 10: Tractable Queries (PostgreSQL)

db	avg(ex_time) msec.	avg(out_time) msec.	avg(res_size) msec.	qmpH	#(triples)
NPD	61	36	$2.3 * 10^4$	5278	≈2M
NPD2	121	71	$4.2 * 10^4$	2684	≈6M
NPD5	173	99	$7.1 * 10^4$	1893	≈12M
NPD10	222	138	$1.1 * 10^5$	1429	≈25M
NPD50	592	355	$2.7 * 10^5$	542	≈116M
NPD100	1066	516	$4.1 * 10^5$	325	≈220M
NPD500	$4.1 * 10^4$	467	$3.3 * 10^5$	12	≈1.3B
NPD1500	$2.6 * 10^5$	3470	$1.15 * 10^6$	1.9	≈4B

actual virtual growth is more evident. Nevertheless, VIG performs significantly better than a purely random approach (one order of magnitude for object properties, 2-3 times for data properties). We shall see how this difference strongly affects the results of the benchmark (Section 6).

5.3 Related Work on Data Generation

There are several data generators that come with database and semantic web benchmarks [3, 20, 12] (see also Footnote 5). As explained before, it is not trivial to re-use a semantic web triple generator (e.g., [30]) since in OBDA this implies solving the view update problem. Therefore, we focus on DB data generators. To the best of our knowledge, most of them (such as the ones for TPC or Wisconsin) are tailored to a given DB schema, and moreover such schemas are rather simple (often around 20 tables). A notable example is the TPC-DS data generator. TPC-DS is the latest TPC benchmark with a scaling, correlation, and skew methodology in the data generator (MUDD) [27]. MUDD takes the distribution of each pair column/table as a manually predefined input, and generates data according to the defined distribution. On the other hand, VIG, collects different statistics about each table, and generates data to keep the statistics constant. MUDD allows a more sophisticate and precise data generation, but it requires a deep understanding of the dataset, and manual settings that can be challenging as the complexity of the schema increases (the TPC-DS schema contains 20 tables, whereas the NPD schema contains 70 tables). We plan to extend our work with distribution analysis so as to replicate the skew that is usually present in real-world data. However, observe that in general skew might not be a crucial factor in determining the shape of virtual instances since repeated triples are removed in the virtual RDF graphs.

6. BENCHMARK RESULTS

We ran the benchmark on the *Ontop* system[13] [23, 16], which, to the best of our knowledge, is the only *fully* implemented OBDA system that is freely available. In addition, we tried a closed OBDA system, *Mastro* [5], that is used in large industrial projects, and two systems that use mappings but that do not provide reasoning, namely OpenLink Virtuoso Views[14] and Morph[15]. Unfortunately,

[13]`http://ontop.inf.unibz.it/`
[14]`http://virtuoso.openlinksw.com/`
[15]`https://github.com/oeg-upm/morph-rdb/`

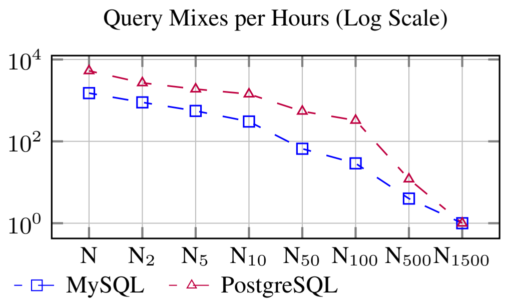


Figure 1: Full summary of Ontop-MySQL vs Ontop-PostgreSQL

Mastro and Virtuoso do not fully support R2RML mappings and Morph is not able to load the mappings for NPD. Ultrawrap [25] is another commercial OBDA system but we were not granted the right to test it.

In order to provide a meaningful comparison, we looked for a triple store that allows for OWL 2 QL reasoning through query rewriting—Virtuoso does not provide this feature. Thus, we compared *Ontop* with *Stardog 2.1.3*. Stardog[16] is a commercial RDF database developed by Clark&Parsia that supports SPARQL 1.1 queries and OWL 2 for reasoning.

Since Stardog is a triple store, we needed to materialize the virtual RDF graph exposed by the mappings and the database using *Ontop*. For the aggregate queries we used an experimental unreleased version of *Ontop* (V2.0) that does not support existential reasoning in conjunction with aggregates.

MySQL and PostgreSQL were used as underlying relational database systems. The hardware consisted of an HP Proliant server with 24 Intel Xeon X5690 CPUs (144 cores @3.47GHz), 106 GB of RAM and a 1 TB 15K RPM HD. The OS is Ubuntu 12.04 LTS. Due to space constraints, we present the results for only one running client. We obtained results with *existential reasoning* turned on (for non-aggregate queries) and off.

In order to test the scalability of the systems w.r.t. the growth of the database, we used the data generator described in Section 5.1 and produced several databases, the largest being approximately 1500 times bigger than the original one ("NPD1500" in Table 9, ≈117 GB of size on disk).

[16]`http://stardog.com/`

Table 12: Hard queries-*Ontop*/MySQL

query	NPD rp_t/weight R+U (sec./ratio)	NPD2 rp_t/weight R+U (sec./ratio)	NPD5 rp_t/weight R+U (sec./ratio)	NPD10 rp_t/weight R+U (sec./ratio)	NPD10 RAND rp_t/weight R+U (sec./ratio)
No Existential Reasoning					
q6	1.5/0.07	8.2/0.02	23/<0.01	51/<0.01	54/<0.01
q9	0.6/0.17	2.3/0.03	4/0.03	50/<0.01	51/<0.01
q10	0.07/0.14	0.1/0.1	0.16/0.06	0.2/0.05	0.3/0.03
q11	0.9/0.1	36/<0.01	198/<0.01	1670/<0.01	70/<0.01
q12	0.8/0.16	41/<0.01	275/<0.01	1998/<0.01	598/<0.01
q13	0.1/0.01	0.2/<0.01	0.2/0.01	0.4/<0.01	1.1/<0.01
q14	0.3/<0.01	1.0/<0.01	1.4/<0.01	0.6/<0.01	5.3/<0.01
q15	1015.7/<0.01	—	—	—	—
q16	1.4/<0.01	13.2/<0.01	32.7/<0.01	171.9/<0.01	406.1/<0.01
q17	—	—	—	—	—
q18	—	—	—	—	—
q19	—	—	—	—	—
q20	211.1/<0.01	1913.4/<0.01	—	—	—
q21	210.9/<0.01	1905.6/<0.01	—	—	—
Existential Reasoning					
q6	8.5/0.35	18/0.19	36/0.09	85/0.04	88/0.03
q9	0.2/0.2	0.2/0.2	0.2/0.2	0.2/0.2	0.2/0.2
q10	0.1/0.2	0.1/0.2	0.3/0.07	0.7/0.03	1.8/0.01
q11	3/0.2	25/0.03	980/<0.01	980/<0.01	41/0.02
q12	686/0.97	733/0.91	868/0.74	2650/0.24	880/0.74

Table 11: Hard Queries Rewriting And Unfolding

query	#rw	#un	rw time sec.	un time sec.
	Ext. Reasoning OFF			
q6	1	48	0	0.1
q9	1	570	0	0.1
q10	1	24	0	0.9
q11	1	24	0	0.1
q12	1	48	0	0.2
q13	1	4	0	0.005
q14	1	2	0	0.01
q15	1	4	0	0.03
q16	1	26	0	0.05
q17	1	40	0	0.1
q18	1	38	0	0.2
q19	1	40	0	0.1
q20	1	13	0	0.04
q21	1	13	0	0.06
	Ext. Reasoning ON			
q6	73	1740	1.8	1.3
q9	1	150	0	0.03
q10	1	24	0	0.01
q11	73	870	0.03	0.7
q12	10658	5220	525	139

Tables 9, 10, and Figure 1 show 7 queries from the initial query set, for which the unfolding produces a single select-project-join (SPJ) SQL query after being *optimised* by *Ontop*. Such optimisations remove redundant self-joins, redundant unions, push joins into unions, etc. (see [24] for a complete description). These results show the scalability of this approach. The query mix of 7 queries was executed 10 times (in each dataset, NPD1–NPD1500), each time with different filter conditions so that the effect of caching is minimized, and statistics were collected in each execution. We measured the sum of the *query execution time* (avg(ex_time)), the time spent by the system to display the results to the user (avg(out_time)), the number of results (avg(res_size)), and the query mixes per hour (qmpH), that is, the number of times that these 7 queries can be answered in one hour. In this experiment we can see that *Ontop*-PostgreSQL runs orders of magnitude faster than *Ontop*-MySQL whenever the query does not contain Optionals. However, for queries that contain optionals, MySQL performs much better. By looking at the query plans[17] in both DB engines, we found out that MySQL can better optimise the query by eliminating left joins over the same table.

For instance, the SQL translation of query 14 requires 2 left joins over the same table. PostgreSQL materialises the subqueries and then performs both left joins. MySQL, on the other hand, can avoid such redundant left joins over the same table.

Table 11 contains results showing the number of unions of SPJ queries generated after rewriting (#rw) and after unfolding (#un) for the 5 hardest queries. In addition, it shows the time spent by *Ontop* on rewriting and unfolding. Here we can observe how existential reasoning can produce a noticeable performance overhead, by producing queries consisting of unions of more than 5000 subqueries (c.f., q12). This blow-up is due to the combination of rich hierarchies, existentials, and mappings. These queries are meant to be used in future research on query optimization in OBDA.

Tables 12 and 13 contain results for the 13 hardest queries in *Ontop*. Some of these queries take hours to be executed, therefore qmpH is not so informative in this case. Thus, we run each query twice with a timeout of 2 hours on the response time. The dashes in the tables represent timeouts. Observe that the response time tends to grow faster than the growth of the underlying database. This follows from the complexity of the queries produced by the unfolding step, which usually contain several joins (remember that the worst case cardinality of a result set produced by a join is quadratic in the size of the original tables). Column NPD10 RAND witnesses how using a purely random data generator gives rise to datasets for which the queries are much simpler to evaluate. This is mainly due to the fact that a random generation of values tends to decrease the ratio of duplicates inside columns, resulting in smaller join results over the tables [28]. Hence, purely randomly generated datasets are not appropriate for benchmarking.

In Figure 2, we compare the response times in *Ontop* and Stardog. As expected, the queries with worst performance in OBDA (q_6, q_9, q_{10},... etc.) are those that were affected by the blow-up shown in Table 11. In this case, Stardog performs orders of mag-

[17]Available in `http://www.inf.unibz.it/~dlanti/techreportNPD-EDBT.pdf`

Table 13: Hard queries-*Ontop*/PostgreSQL

query	NPD rp_t/weight R+U (sec./ratio)	NPD2 rp_t/weight R+U (sec./ratio)	NPD5 rp_t/weight R+U (sec./ratio)	NPD10 rp_t/weight R+U (sec./ratio)	NPD10 RAND rp_t/weight R+U (sec./ratio)
No Existential Reasoning					
q06	1.2/0.15	1.3/0.11	3.9/0.04	4.3/0.03	3.6/0.03
q09	6.4/0.28	6.0/0.21	7.5/0.32	8.6/0.19	1.5/0.1
q10	0.2/0.16	0.6/0.03	0.7/0.03	0.9/0.03	1.4/0.01
q11	1.1/0.12	22.1/<0.01	66.1/<0.01	160.3/<0.01	110.9/<0.01
q12	1.9/0.16	20.9/0.01	101.6/<0.01	195.6/<0.01	121.5/<0.01
q13	0.9/0.06	0.2/0.02	0.5/<0.01	0.4/<0.01	0.7/<0.01
q14	453.2/<0.01	—	—	—	—
q15	122.9/<0.01	366.3/<0.01	771.3/<0.01	1491.2/<0.01	1296.9/<0.01
q16	1.5/0.01	17.0/<0.01	64.6/<0.01	237.8/<0.01	588.6/<0.01
q17	—	—	—	—	—
q18	—	—	—	—	—
q19	—	—	—	—	—
q20	1.8/<0.01	5.2/<0.01	10.5/<0.01	20.0/<0.01	19.2/<0.01
q21	1.8/<0.01	5.2/<0.01	10.5/<0.01	19.7/<0.01	12.2/<0.01
Existential Reasoning					
q06	14.9/0.14	48.6/0.04	52.0/0.07	55.0/0.02	58.0/0.01
q09	0.4/0.16	0.3/0.15	0.7/0.1	1.0/0.1	0.5/0.1
q10	0.2/0.14	0.5/0.07	0.6/0.04	0.8/0.06	1.4/0.02
q11	17.5/0.08	117.3/<0.01	—	—	—
q12	1395.6/0.5	4090.8/0.47	—	—	—

nitude faster than *Ontop*. These queries should guide the future research in query optimisation in OBDA. On the other hand, the queries that perform well (q_1, q_2, q_3,... etc.) are those where the different optimizations lead to a simple SPJ SQL query. Note that the times required to materialize (by *Ontop*) and load the dataset in Stardog go from 1 min. (NPD1) to 1 hour (NPD10).

7. CONCLUSIONS AND FUTURE WORK

The benchmark proposed in this work is the first one that thoroughly analyzes a complete OBDA system implementation in all significant components, including query rewriting, query unfolding, and query execution. So far, little or no work has been done in this direction, as pointed out in [19]. This benchmark reveals the strengths and pitfalls of OBDA. We confirmed that this approach can be orders of magnitude faster than standard triple stores, fully exploiting the highly optimized DB engines. To achieve such performance, structural and semantic optimizations of the SQL translation are required. However, the results also show that the exponential blowup in the unfolding phase is a major source of performance loss of modern OBDA systems. If this issue is not handled properly, it can prevent OBDA systems from being deployed in production environments. This explosion, however, can be strongly reduced using tuning and optimisation techniques that exploit the information hidden in the data, such as functional dependencies, redundant mappings, etc. We are currently working on this topic.

For a better analysis it is crucial to refine the generator in such a way that domain-specific information is taken into account, and a better approximation of real-world data is produced.

Acknowledgement. This paper is supported by the EU under the large-scale integrating project (IP) Optique (Scalable End-user Access to Big Data), grant agreement n. FP7-318338.

8. REFERENCES

[1] Abadi, D., et al.: The Beckman report on database research (2013), available at `http://beckman.cs.wisc.edu/`

[2] Bail, S., Alkiviadous, S., Parsia, B., Workman, D., van Harmelen, M., Goncalves, R.S., Garilao, C.: FishMark: A linked data application benchmark. In: Proc. of the Joint Workshop on Scalable and High-Performance Semantic Web Systems (SSWS+HPCSW), vol. 943, pp. 1–15. CEUR, `ceur-ws.org` (2012)

[3] Bizer, C., Schultz, A.: The Berlin SPARQL benchmark. Int. J. on Semantic Web and Information Systems 5(2), 1–24 (2009)

[4] Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Poggi, A., Rodríguez-Muro, M., Rosati, R.: Ontologies and databases: The *DL-Lite* approach. In: RW Tutorial Lectures, LNCS, vol. 5689, pp. 255–356. Springer (2009)

[5] Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Poggi, A., Rodriguez-Muro, M., Rosati, R., Ruzzi, M., Savo, D.F.: The Mastro system for ontology-based data access. Semantic Web J. 2(1), 43–53 (2011)

[6] Calvanese, D., Lanti, D., Rezk, M., Slusnys, M., Xiao, G.: A scalable benchmark for OBDA systems: Preliminary report. In: Proc. of ORE. CEUR, `ceur-ws.org` (2014)

[7] Cosmadakis, S.S., Papadimitriou, C.H.: Updates of relational views. JACM 31(4), 742–760 (1984)

[8] Das, S., Sundara, S., Cyganiak, R.: R2RML: RDB to RDF mapping language. W3C Recommendation, W3C (Sep 2012), available at `http://www.w3.org/TR/r2rml/`

[9] DeWitt, D.J.: The Wisconsin benchmark: Past, present, and future. In: The Benchmark Handbook for Database and Transaction Systems. Morgan Kaufmann, 2 edn. (1993)

[10] Franconi, E., Guagliardo, P.: The view update problem revisited. CoRR Technical Report arXiv:1211.3016, arXiv.org e-Print archive (2012), available at `http://arxiv.org/abs/1211.3016`

[11] Giese, M., Haase, P., Jiménez-Ruiz, E., Lanti, D., Özçep, Ö., Rezk, M., Rosati, R., Soylu, A., Vega-Gorgojo, G., Waaler, A., Xiao, G.: Optique – Zooming in on Big Data access. IEEE Computer (2015)

[12] Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. J. of Web Semantics 3(2–3), 158–182 (2005)

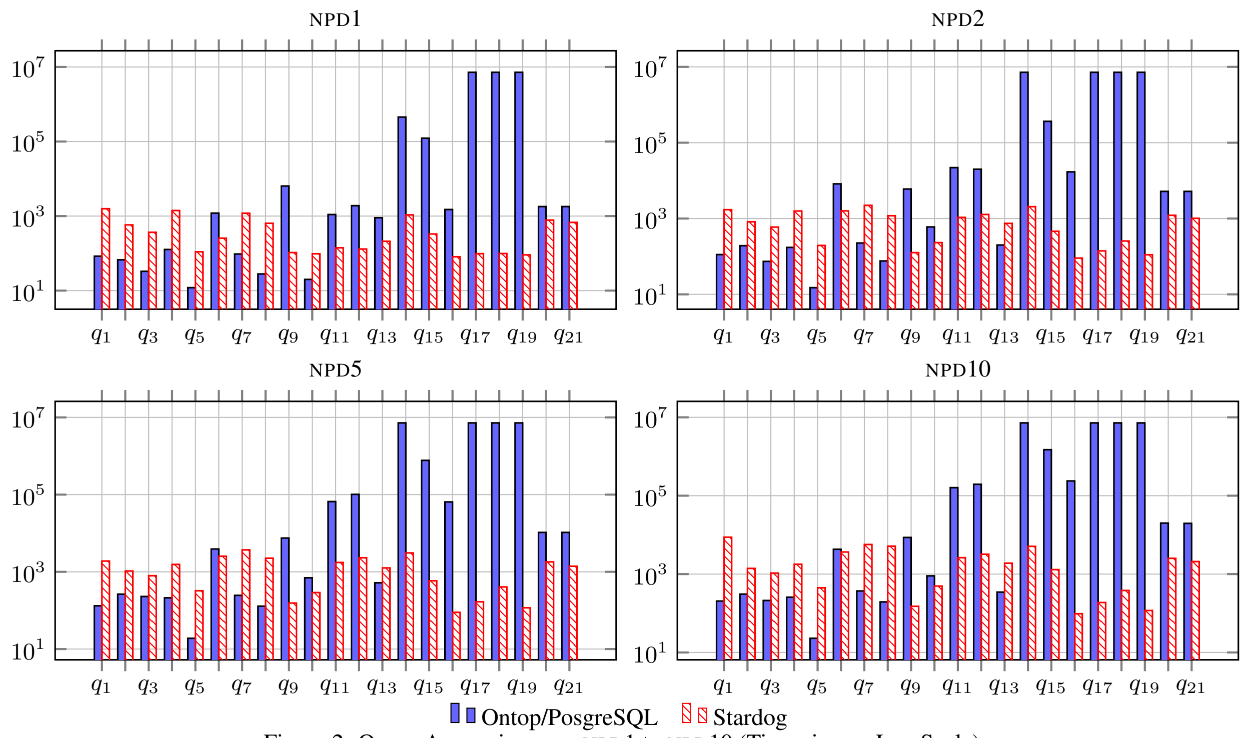


Figure 2: Query Answering over NPD1 to NPD10 (Times in ms, Log Scale)

[13] Guo, Y., Qasem, A., Pan, Z., Heflin, J.: A requirements driven framework for benchmarking semantic web knowledge base systems. IEEE TKDE 19(2), 297–309 (2007)

[14] Keet, C.M., Alberts, R., Gerber, A., Chimamiwa, G.: Enhancing web portals with Ontology-Based Data Access: the case study of South Africa's Accessibility Portal for people with disabilities. In: Proc. of OWLED. CEUR, `ceur-ws.org`, vol. 432 (2008)

[15] Kontchakov, R., Lutz, C., Toman, D., Wolter, F., Zakharyaschev, M.: The combined approach to query answering in *DL-Lite*. In: Proc. of KR. pp. 247–257 (2010)

[16] Kontchakov, R., Rezk, M., Rodriguez-Muro, M., Xiao, G., Zakharyaschev, M.: Ontology-based data access: Ontop of databases. In: Proc. of ISWC. LNCS, vol. 8218, pp. 558–573. Springer (2013)

[17] Lanti, D., Rezk, M., Slusnys, M., Xiao, G., Calvanese, D.: The NPD benchmark for OBDA systems. In: Proc. of SSWS. CEUR, `ceur-ws.org` (2014)

[18] LePendu, P., Noy, N.F., Jonquet, C., Alexander, P.R., Shah, N.H., Musen, M.A.: Optimize first, buy later: Analyzing metrics to ramp-up very large knowledge bases. In: Proc. of ISWC. LNCS, vol. 6496, pp. 486–501. Springer (2010)

[19] Mora, J., Corcho, O.: Towards a systematic benchmarking of ontology-based query rewriting systems. In: Proc. of ISWC. LNCS, vol. 8218, pp. 369–384. Springer (2013)

[20] Morsey, M., Lehmann, J., Auer, S., Ngonga Ngomo, A.C.: DBpedia SPARQL Benchmark – Performance assessment with real queries on real data. In: Proc. of ISWC, Volume 1. LNCS, vol. 7031, pp. 454–469. Springer (2011)

[21] Poggi, A., Lembo, D., Calvanese, D., De Giacomo, G., Lenzerini, M., Rosati, R.: Linking data to ontologies. J. on Data Semantics X, 133–173 (2008)

[22] Rodríguez-Muro, M., Calvanese, D.: Dependencies: Making ontology based data access work in practice. In: Proc. of AMW. CEUR, `ceur-ws.org`, vol. 749 (2011)

[23] Rodriguez-Muro, M., Kontchakov, R., Zakharyaschev, M.: Answering SPARQL queries over databases under OWL 2 QL entailment regime. In: Proc. of ISWC. LNCS, vol. 8218. Springer (2014)

[24] Rodriguez-Muro, M., Rezk, M.: Efficient SPARQL-to-SQL with R2RML mappings (2014), submitted for publication

[25] Sequeda, J.F., Miranker, D.P.: Ultrawrap: SPARQL execution on relational data. J. of Web Semantics 22, 19–39 (2013)

[26] Skjæveland, M.G., Lian, E.H., Horrocks, I.: Publishing the Norwegian Petroleum Directorate's FactPages as Semantic Web data. In: Proc. of ISWC. LNCS, vol. 8219, pp. 162–177. Springer (2013)

[27] Stephens, J.M., Poess, M.: MUDD: A multi-dimensional data generator. In: Proc. of the 4th Int. Workshop on Software and Performance. pp. 104–109. ACM (2004)

[28] Swami, A., Schiefer, K.B.: On the estimation of join result sizes. In: Proc. of EDBT. LNCS, vol. 779, pp. 287–300. Springer (1994)

[29] Venetis, T., Stoilos, G., Stamou, G.B.: Query extensions and incremental query rewriting for OWL 2 QL ontologies. J. on Data Semantics 3(1), 1–23 (2014)

[30] Wang, S.Y., Guo, Y., Qasem, A., Heflin, J.: Rapid benchmarking for semantic web knowledge base systems. In: Proc. of ISWC. LNCS, vol. 3729, pp. 758–772. Springer (2005)

[31] Weithöner, T., Liebig, T., Luther, M., Böhm, S.: What's wrong with OWL benchmarks. In: Proc. of the 2nd Int. Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS). pp. 101–114 (2006)

Event Recognition for Maritime Surveillance

Kostas Patroumpas[1,2], Alexander Artikis[3,4], Nikos Katzouris[4],
Marios Vodas[1], Yannis Theodoridis[1], Nikos Pelekis[5]
[1]Department of Informatics, University of Piraeus, Greece
[2]School of Electrical & Computer Engineering, National Technical University of Athens, Greece
[3]Department of Maritime Studies, University of Piraeus, Greece
[4]Institute of Informatics & Telecommunications, NCSR Demokritos, Athens, Greece
[5]Department of Statistics & Insurance Science, University of Piraeus, Greece
{kpatro, a.artikis, mvodas, ytheod, npelekis}@unipi.gr, nkatz@iit.demokritos.gr

ABSTRACT

We present a system that combines intelligent online tracking with complex event recognition against streaming positions relayed from numerous vessels. Given the vital importance of maritime safety to the environment, the economy, and in national security, our system leverages the real-time acquisition of vessel activity with geographical and other static information. Thus, it can offer timely notification in emergency situations, such as intrusion into marine preservation areas, loitering, and unsafe sailing. Thanks to a mobility tracking module, evolving trajectories generated by massive positional updates can be compressed online into concise, but reliable synopses per ship, retaining only salient motion features within a sliding window. These features are exploited by a complex event recognition module that detects suspicious situations of interest to maritime authorities. We conducted a comprehensive empirical validation against a real dataset of traces collected from thousands of vessels. Our results confirm the scalability and approximation accuracy of the proposed system, and thus demonstrate its potential for effective, real-time maritime monitoring.

1. INTRODUCTION

Maritime surveillance systems have been attracting attention both for economic and environmental reasons [1, 3, 16]. For instance, preventing ship accidents by monitoring vessel activity represents substantial savings in financial cost for shipping companies (e.g., oil spill cleanup) and averts irrevocable damages to maritime ecosystems (e.g., fishery closure). Nowadays, maritime navigation technology can automatically provide real-time information from sailing vessels. The Automatic Identification System (AIS) [24] is a tracking system for identifying and locating vessels at sea through data exchange: either with other ships nearby, or AIS base stations along coastlines, or even satellites when out of range of terrestrial networks. AIS is intended to assist vessel crews in collision avoidance and allows maritime authorities to monitor vessel movements. This technology integrates a VHF transceiver with a positioning device (e.g., GPS), and other electronic navigation sensors, such as a gyrocompass or rate of turn indicator.

AIS raw tracking data offers a wealth of information including unique identification of vessels, their position, course, and speed. Such information can be displayed on screen aboard of the ship or in maritime control centers. AIS-equipped vessels may be of diverse type, size, or tonnage. Not all of them relay their position simultaneously or at a fixed frequency, but depending on the transponder configuration aboard the ship, proximity to base stations, and the type of their motion. Vessels anchored or slowly moving transmit less frequently than those cruising fast in the open sea or manoeuvering near the docks. Note that this data is not noise-free; AIS messages may be delayed, intermittent, or conflicting.

Considering that AIS information is continuously emitted from over 400 thousand ships worldwide [2], it evidently fulfills all four '*V*' challenges (*Volume, Velocity, Variety, lack of Veracity*), as well as the '*D*' challenge (*Distribution of data sources*) in big data management. Therefore, for effective vessel identification and tracking, maritime surveillance systems need to scale to the increasing traffic activity witnessed in the past few years[1]. Such systems should detect threats and abnormal activity over voluminous, fluctuating, and noisy data streams from thousands of vessels, and also correlate them with static data expressing vessel characteristics (type, tonnage, cargo, etc.) and geographical information (such as bathymetric data and protected areas).

To address these requirements, we introduce a maritime surveillance system that consists of two main components. A *trajectory detection* component consumes a positional stream of AIS messages from a large fleet and tracks major changes along each vessel's movement. Thus, it can instantly identify "critical points" en route, indicating important changes like a stop, a sudden turn, or slow motion of a ship. Except for harsh weather conditions, traffic regulations, local manoeuvres in ports, etc., ships are expected to move along almost straight, predictable paths. Therefore, most of the frequently relayed positional messages are not really required for representing the actual trace of a vessel. Instead, by discarding superfluous locations along a "normal" course with a known velocity, we can approximately reconstruct each vessel's *trajectory* from the sequence of its critical points only. This online summarization achieves data compression close to 95%, incurring negligible loss in approximation accuracy. With such dramatic reduction in system load, execution of continuous and historical queries can be greatly improved, e.g., reducing latency of online collision detection or similarity search among recent vessel paths.

The detected critical points may be used for map display, but they are mostly valuable for recognition of complex phenomena and thus issuing alert notifications to marine authorities. To this end, a *complex event recognition* component combines the derived

[1]`http://www.bbc.com/news/science-environment-28372461`

10.5441/002/edbt.2015.63

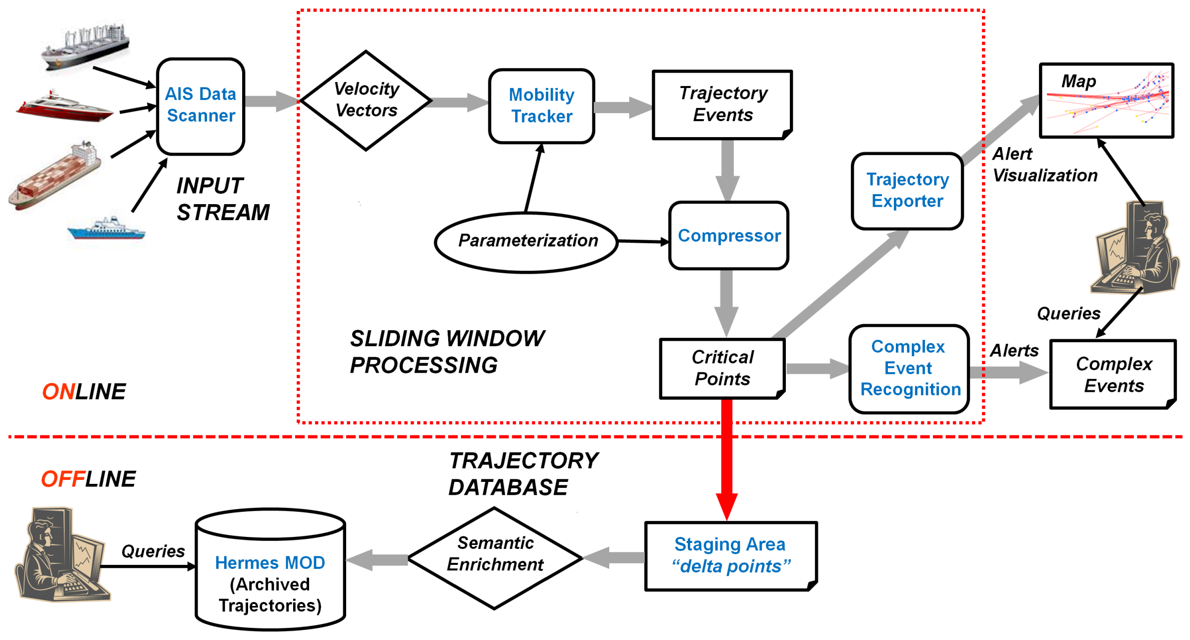


Figure 1: Processing scheme of the maritime surveillance system.

stream of critical points expressing vessel activity, with static geographical and vessel data, and can detect suspicious or potentially dangerous situations, such as loitering, vessels passing through protected areas, and unsafe shipping. In contrast to map display of current locations or information about specific vessels of interest [2], this module can be used to spot emergency situations in real-time.

The recognized complex events and lightweight trajectory synopses may be physically archived in a database for extracting *offline analytics*, including travel statistics, motion trends, origin–destination matrices, frequent routes, area and vessel classification (suspicious areas, illegal or dangerous shipping), and much more.

In this paper, we emphasize the real-time features of our maritime surveillance system, developed in the context of the AMINESS project [1]. This interdisciplinary project aims to reinforce safety and assist in the management of sea environments, particularly in the Aegean Sea. We conducted an extensive empirical validation of the proposed system on a large dataset of real vessel traces collected in the summer of 2009 from AIS base stations along the coastline of Greece. Our study confirms that the high compression ratio achieved by the trajectory detection component, along with the efficient pattern matching algorithms of the complex event recognition component, allow this system to scale to high velocity data streams expressing the current activity of large fleets.

The remainder of the paper is organized as follows. In Section 2, we present the architecture of the proposed maritime surveillance system. Sections 3 and 4 respectively present the two main components: the trajectory detection and complex event recognition modules. Section 5 reports performance results from a comprehensive evaluation of the implemented system. In Section 6, we compare our approach against related systems and methodologies. Finally, in Section 7 we summarize our work and outline directions for further research and implementation.

2. SYSTEM ARCHITECTURE

In this Section, we outline the processing flow of the proposed maritime surveillance system. As illustrated in Figure 1, the system consumes a stream of AIS tracking messages from vessels, detects important features that characterize their movement and recognizes complex events such as suspicious vessel activity. These results can be used to evaluate *continuous location-aware queries*, e.g., to detect whether a ship is approaching a port or if a vessel has just entered into an environmentally protected area.

In order to meet the real-time requirements of data stream processing, this online process necessitates the use of a *sliding window* [18, 29], which abstracts the time period of interest and keeps up with the evolving movement. Typically, a window looks for phenomena that occurred in a recent *range* ω (e.g., positions received during past 60 minutes). This window moves forward to keep in pace with newly arrived stream tuples, so it gets refreshed at a specific *slide step* every β units (e.g., each minute). For instance, an aggregate query could report at every minute (β) the distance traveled by a ship over the past hour (ω). Typically, it holds that $\beta < \omega$; so, as time goes by, successive window instantiations may share positional tuples over their partially overlapping ranges.

As input, we consider AIS messages of certain types (1, 2, 3, 18, 19) and extract position reports. Each message specifies the $MMSI$ (Maritime Mobile Service Identity) of the reporting vessel. For a given $MMSI$, each of its successive positional samples p consists of longitude/latitude coordinates (Lon, Lat) measured at discrete, totally ordered timestamps τ (e.g., at the granularity of seconds). Without loss of generality, we abstract vessels as 2-dimensional point entities moving across time, because our primary concern is to capture their motion features. By monitoring the timestamped locations from a large fleet of N vessels, the system must deal with a *positional stream* of tuples $\langle MMSI, Lon, Lat, \tau\rangle$. A *Data Scanner* decodes each AIS message, identifies those four attributes (the rest are ignored in our analysis), and cleans them from distortions caused during transmission (e.g., discard messages with bad checksum). This constitutes an *append-only* data stream, as no deletions or updates are allowed to already received locations.

But it is the sequential nature of each vessel's trace that mostly matters for capturing movement patterns en route (e.g., a slow turn), as well as spatiotemporal interactions (e.g., ships traveling together).

Such a *trajectory* is approximated as an evolving sequence of successive point samples that locate this vessel at distinct timestamps (e.g., every few seconds). In order to detect motion changes, the *Mobility Tracker* module maintains one velocity vector per vessel based on its two most recent positions[2]. Working entirely in main memory and without any index support, the Mobility Tracker checks when and how velocity changes with time. Thus, it can detect trajectory events, either instantaneous (e.g., a sudden turn) or of longer duration (e.g., a smooth turn). At each window slide, those events are properly filtered (e.g., from possible outliers) via a *Compressor*. Then, a sequence of *"critical"* points (such as a stop) is emitted, which are much fewer compared to the originally relayed positions. So, the current vessel motion can be characterized in real time with particular *annotations* (e.g., stop, turn). Once new trajectory events are detected per vessel upon each window slide, the annotated critical points can be readily emitted and visualized on maps through a *Trajectory Exporter*, e.g., as KML polylines (for trajectories) and placemarks (for vessel locations).

Moreover, the derived critical points are transmitted to the *Complex Event Recognition* module, which combines this event stream with static geographical and vessel data, such as bathymetric data and protected areas. The objective of this process is to detect potentially suspicious or dangerous situations, such as loitering, vessels passing through protected areas, and unsafe shipping. The recognized complex events are pushed in real-time to the end user (marine authorities) for real-time decision-making.

Finally, historical information can be progressively compiled from these detected features. "Delta" critical points (issued once the window slides forward) are periodically sent from main memory into a staging area on disk. An *offline* module accepts these lightweight, digested traces and reconstructs trajectory segments for archiving in Hermes Moving Objects Database (MOD) [30], instead of naïvely storing enormous quantities of raw AIS positions. This reconstruction process also identifies ships docked at ports, so that trajectory semantics can be enriched accordingly so as to extract further knowledge through motion mining or computation of statistics.

3. DETECTING TRAJECTORY EVENTS

With the possible exception of local manoeuvres near ports, marine regulations, or harsh weather conditions, vessels are normally expected to follow almost straight, predictable routes. In terms of vessel mobility, what matters most is to detect when and how the general course has changed, e.g., identify a stop, a turning point, or slow motion. Such *trajectory movement events* (ME) suffice to indicate "critical points" along the trace of each vessel and thus offer a concise, yet quite reliable representation of its course. It turns out that a large portion of the raw positional reports can be suppressed with minimal loss in accuracy, as they hardly contribute any additional knowledge. We distinguish two kinds of trajectory events:

- *Instantaneous trajectory events* involve individual time points per route, by simply checking potentially important changes with respect to the previously reported location (e.g., a sharp change in heading).
- *Long-lasting trajectory events* are deduced after examining a sequence of instantaneous events over a longer time period in order to identify evolving motion changes. For example, a few consecutive changes in heading may be very small if each is examined in isolation from the rest, but cumulatively they could signify a notable change in the overall direction.

[2]Typically for trajectories [8], linear interpolation is applied between each pair of successive measurements (p_i, τ_i) and (p_{i+1}, τ_{i+1}). For simplicity, we assume that this also holds in the case of vessels. With the exception of intermittent signals, their course between any two consecutive positions practically evolves in a very small area, which can be locally approximated with a Euclidean plane using Haversine distances.

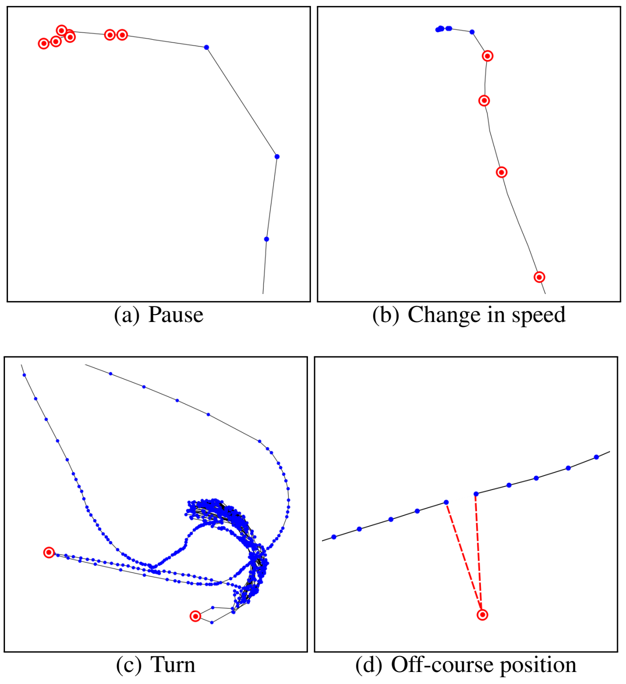

(a) Pause (b) Change in speed (c) Turn (d) Off-course position

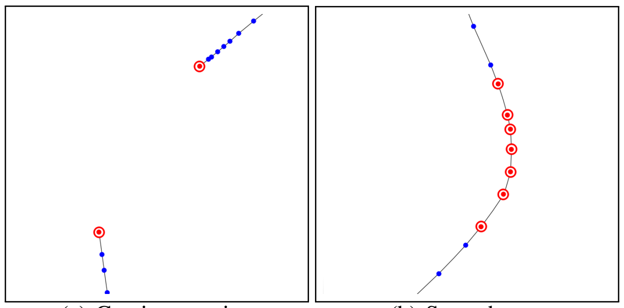
Figure 2: Instantaneous events and outliers in a vessel's course.

In this Section, we first describe how the sequence of vessel positions can be processed *online* in order to detect such trajectory events. An early version of the online tracking module was introduced in [28]. This has been substantially enhanced to identify additional events and much more robust in coping with increased data volumes. We also explain how the resulting critical points per vessel can provide a lightweight summary of its trajectory, which then offers many opportunities for affordable *offline* analysis.

3.1 Online Tracking of Moving Vessels

As illustrated in Figure 1, the system accepts fresh AIS messages from ships and extracts positional tuples $\langle MMSI, Lon, Lat, \tau \rangle$. In order to identify significant changes in movement, it first computes the instantaneous velocity vector $\vec{v}_{now}$ from the two most recent positions reported by each vessel $MMSI$. Then, the *mobility tracker* can instantly deduce a variety of *instantaneous* events by examining the trace of each vessel alone:

- *Pause* indicates whether a vessel is currently halted, once its instantaneous speed $\vec{v}_{now}$ does not exceed a suitable threshold v_{min}. For example, if $\vec{v}_{now}$ is currently less than $v_{min} = 1$ knot, then the ship rests practically immobile. For the vessel shown in Figure 2(a), the red bullets indicate several pause events; apparently, the ship is anchored at the port and such small displacements may be caused by GPS errors or sea drift.
- *Speed change* is issued once current v_{now} deviates by more than $\alpha\%$ from the previously observed speed v_{prev}. For a given threshold α, the formula $|\frac{v_{now}-v_{prev}}{v_{now}}| > \frac{\alpha}{100}$ indicates whether the vessel has just decelerated or accelerated. This is normally the case when a ship is approaching to or departing from a port, as depicted in Figure 2(b).


(a) Gap in reporting (b) Smooth turn

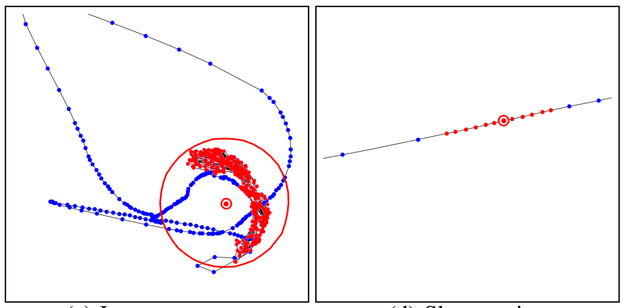

(c) Long-term stop (d) Slow motion

Figure 3: Long-lasting trajectory events.

- *Turn*: It occurs when the direction has changed by more than a given angle $\Delta\theta$, e.g., there is a difference of more than 15^o from its previous heading. Red bullets in Figure 2(c) illustrate two such sudden turns.

- *Off-course positions* incur a very abrupt change in vessel's velocity $\overrightarrow{v}_{now}$ (both in speed and heading). Yet, such an outlier can be easily detected as it signifies an abnormal, yet only temporary, deviation from the known course as abstracted by mean velocity $\overrightarrow{v_m}$ of the ship over its previous m positions. Figure 2(d) illustrates such a case.

No critical point gets immediately issued upon detection of any such simple events. An instantaneous pause or turn may be coincidental and is not meaningful out of context, because a series of such events may signify that the ship is stopped for some time, as we will explain shortly. Besides, accepting an outlier would drastically distort the resulting trajectory representation, as the red dashed line in Figure 2(d) illustrates. Even worse, such noisy positions may affect detection of important events. For instance, an outlier breaking the subsequence of instantaneous pause events could prevent characterization of a long-term stop, and instead yield two successive such stops very close to each other. Thus, any off-course positions should be discarded as noise.

Those instantaneous events are used to detect spatiotemporal phenomena of some duration, i.e., *long-lasting trajectory events* like:

- *Gap* in reporting is issued when a vessel has not emitted a message for a time period ΔT, e.g., over the past 10 minutes. Therefore, its course is unknown during this period, as it occurs between the two red bullets in Figure 3(a). Reporting such a critical point (i.e., when the gap started) is important, not only for properly monitoring vessels, but also for safety reasons, e.g., a suspicious move near maritime boundaries, or a potential intrusion of a tanker into a marine park.

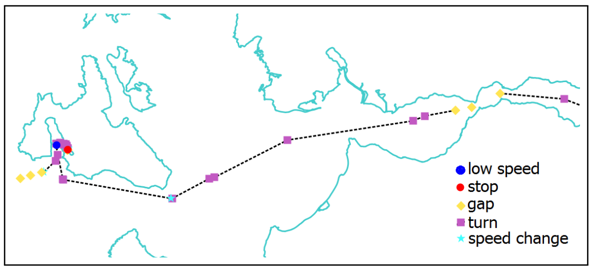


Figure 4: Critical points identified along a vessel trajectory.

- *Smooth turn* can be identified by checking whether the cumulative change in heading across a series of previous positions exceeds a given angle $\Delta\theta$, as illustrated with the red points in Figure 3(b). Then, the latest of them is emitted as a critical (turning) point.

- *Long-term stop* occurs when at least m consecutive instantaneous pause or turn events are found within a predefined radius r (e.g., 200 meters). In Figure 3(c), the red points inside the circle succeed one another and indicate such immobility, so they could be collectively approximated by a single critical point (their centroid) with their total duration.

- *Slow motion* means that the vessel consistently moves at low speed ($\leq v_{min}$) over its m last messages, as in Figure 3(d). In contrast to a stop event, these successive locations usually occur along a path and not all of them fall in a small circle. The median of these m positions is reported as a representative critical point.

Thus, critical points are emitted from each long-lasting event. Provided that they do not qualify for outliers, instantaneous events for speed change (Figure 2(b)) or isolated turns (Figure 2(c)) also contribute to critical points. The example trajectory in Figure 4 illustrates the data compression gains achieved when retaining critical points only. Obviously, such filtering greatly depends on proper choice of parameter values, which is a trade-off between reduction efficiency and approximation accuracy. For a suitable calibration of these parameters, apart from consulting maritime domain experts (our partners in the AMINESS project [1]), we have also conducted several exploratory tests on randomly chosen vessels from AIS data in the Aegean Sea. For instance, setting $\Delta\theta = 5^o$ instead of $\Delta\theta = 15^o$ incurs a 10% increase in the amount of critical points, because more original AIS locations would qualify as turning points due to sea drift and discrepancies in GPS signals. Since our analysis is mostly geared towards data reduction, for our empirical study (Section 5.1) we have chosen the aggressive parametrization listed in Table 3, which yields quite tolerable accuracy. With relaxed parameter values, additional events can be detected, capturing slighter changes in each trajectory.

The complexity for detecting instantaneous events and communication gaps is $O(1)$ per incoming positional tuple, since only the two latest locations are examined per vessel. The cost for the remaining long-lasting events is $O(m)$, where m is the number of latest positions that need inspection. As m is usually a small integer (we set $m = 10$ in our experiments), this cost is affordable.

Rules for such trajectory events are suitably defined in the mobility tracker, which is equipped with robust data structures for in-memory maintenance of movement features. Note that more events can be detected by simply enhancing the mobility tracker with extra conditions. In future work, we plan to complement this methodology so as to capture additional features, such as traveled distance from a given origin (e.g., a port). Better coping with noisy situations is also a challenge, e.g., ignoring *delayed positions* that erroneously mark a ship moving back and forth along its course. Nonetheless, even with this set of events, we can figure out the mutability in each trajectory and distinctly characterize its course across time. Most importantly, these spatiotemporal features can serve as a basis to recognize more complex maritime events, as we discuss later in Section 4.

3.2 Trajectory Reconstruction

By taking advantage of those online annotations at critical points along trajectories, lightweight, succinct *synopses* can be retained per vessel over the recent past. Then, the compressor evicts point locations that have not been detected as critical. Instead of resorting to a costly simplification algorithm, we opt to reconstruct vessel traces approximately from already available critical points. This summarization depends on the type of detected movement events (i.e., stop, low speed, turn, gap, speed change), so as to refresh each trajectory accordingly. This main-memory process affects trajectory portions currently within the sliding window.

But we also incrementally update trajectories in the underlying database with *"delta" points*. Once the window slides forward, expiring critical points are transferred in an intermediate *staging table* on disk. So, this table temporarily records all recent "delta" changes, i.e., critical points evicted from the window, but not yet admitted in disk-based trajectories. Obviously, information archived in the database is always lagged behind the current vessel reports by ω. This is a deliberate decision dictated for data consistency reasons, so as to avoid having any trajectory portions duplicated in memory (online) and on disk (offline).

Offline trajectory reconstruction in Hermes MOD [30] periodically converts a sequence of critical points per ship into disjoint, consecutive trajectory segments. In particular, a long journey breaks up into smaller trips between ports. This segmentation reduces the latency for offline queries in the database, and also decreases the cost of trajectory updates on every batch of critical points. Eventually, instead of representing the entire motion of a vessel with one long trajectory that gets repetitively updated, Hermes MOD deals with multiple, but much smaller segments; only the last segment per vessel may receive any updates.

AIS messages sometimes include information regarding the destination of sailing vessels. Unfortunately, after scrutinizing AIS data samples, we concluded that this voyage-related information is often missing or error-prone, mainly because it is updated manually by the crew. So, we employ an automated procedure for performing *semantic enrichment* of trajectories with added value information about trips between ports. This method takes as input the critical points identified as long-term stops and a set of known port areas (polygons). Once a stop is located inside such a polygon, the name of the respective port becomes an attribute of that point. It is reasonable to assume that between two such distinct stops O and D, the ship sailed from origin port O and reached destination port D. Then, this is identified as a new trip with a known destination D. Note that origin port O may remain unknown, because the ship might have been on the move when the AIS base stations started receiving its signals. Of course, as each vessel continues sailing, more and more critical points will be detected. However, as long as a specific destination port is not yet identified, these points will be piling up in the staging table awaiting assignment to a trajectory.

3.3 Offline Trajectory Analysis

Once reconstructed trajectories get stored in Hermes MOD, useful statistics and patterns can be extracted from them in an *offline* fashion. Since the focus of this paper is on online processing, we only give a brief overview of such analytics. First, a series of *derived tables* can offer historical information about traveled distances and travel times per ship, idle periods at dock, visited ports, etc. Such aggregates may be obtained at various time granularities (e.g., per week, month, or year) and may be computed by other dimensions as well (e.g., flag, cargo, vessel type, etc.). By maintaining *Origin-Destination matrices*, we may identify connections between ports and compute aggregated statistics (duration, speed, frequency, etc.) for such itineraries, often varying by ship type, period of the year, etc. In addition, *motion patterns* can be identified, such as frequently traveled paths ("corridors"), periodicity in movement (e.g., ferries between two specific ports), etc. Hermes MOD incorporates an algorithm for *spatiotemporal clustering*, which can help exploring periodicity of trips. Indeed, two (or more) trajectory clusters may be almost identical spatially, but they are distinct because the temporal dimension is taken into consideration when calculating distances between pairs of trajectory segments.

4. COMPLEX EVENT RECOGNITION

The critical Movement Events (ME) computed by the trajectory detection component are transmitted to the *Complex Event Recognition* module. This module correlates the derived stream of MEs with static geographical and vessel information, such as bathymetric data and locations of protected areas, to detect potentially suspicious and dangerous situations, such as forbidden fishing or unsafe shipping. When recognized, such Complex Events (CE) are forwarded to the marine authorities for real-time decision making.

Our CE recognition component is based on the Event Calculus for Run-Time reasoning (RTEC) [5, 6]. The Event Calculus [17] is a logic programming language for representing and reasoning about events and their effects. The benefits of a logic programming approach to CE recognition are well-documented [26]: such an approach has a formal, declarative semantics, and direct routes to machine learning for constructing CE definitions in an automated way. The use of the Event Calculus has additional advantages: the process of CE definition development is considerably facilitated, as the Event Calculus includes built-in rules for complex temporal representation and reasoning, including the formalization of *inertia*. With the use of the Event Calculus, one may develop intuitive, succinct CE definitions, facilitating the interaction between CE definition developer and domain expert (marine authorities), and allowing for code maintenance. In this Section, we present RTEC following [5, 6], and illustrate its use for maritime surveillance.

4.1 Representing Maritime Activities

The time model of RTEC is linear and includes integer time-points (such as the timestamps of the MEs computed by the trajectory event detection component). Variables start with an upper-case letter, while predicates and constants start with a lower-case letter. Where F is a *fluent*—a property that is allowed to have different values at different points in time—the term $F = V$ denotes that fluent F has value V. Boolean fluents are a special case in which the possible values are true and false. holdsAt$(F = V, T)$ represents that fluent F has value V at a particular time-point T. holdsFor$(F = V, I)$ represents that I is the list of the maximal intervals for which $F = V$ holds continuously. holdsAt and holdsFor are defined in such a way that, for any fluent F, holdsAt$(F = V, T)$ if and only if T belongs to one of the maximal intervals of I for which holdsFor$(F = V, I)$.

The happensAt predicate represents an instance of an event type. E.g., happensAt$(turn(vessel_1), 5)$ represents the occurrence of event type $turn(vessel_1)$ at time 5. When it is clear from the context, we do not distinguish between an event and its type. An *event description* in RTEC includes rules that define the event instances with the use of the happensAt predicate, the effects of events with the use of the initiatedAt and terminatedAt predicates, and the values of the fluents with the use of the holdsAt and holdsFor predicates, as well as other, possibly atemporal, constraints. Table 1 presents the predicates available to the event description developer.

We represent instantaneous MEs and CEs by means of happensAt, while durative MEs and CEs are represented as fluents. In the maritime surveillance setting, the stream of critical MEs, which constitutes the input of RTEC, consists of both instantaneous MEs, such as $speedChange(Vessel)$, and durative ones, such as $stopped(Vessel) = \mathsf{true}$ indicating the maximal intervals during which a $Vessel$ is considered stopped. The majority of CEs are durative and, therefore, in CE recognition the task generally is to compute the maximal intervals for which a fluent representing a CE has a particular value continuously.

For a fluent F, $F = V$ holds at a particular time-point T if $F = V$ has been *initiated* by an event that has occurred at some time-point earlier than T, and has not been *terminated* at some other time-point in the meantime. This is an implementation of the *law of inertia*. To compute the *intervals* I for which $F = V$, that is, $\mathsf{holdsFor}(F = V, I)$, we find all time-points T_s at which $F = V$ is initiated, and then, for each T_s, we compute the first time-point T_f after T_s at which $F = V$ is 'broken'. The time-points at which $F = V$ is initiated are computed by means of domain-specific initiatedAt rules. The time-points at which $F = V$ is 'broken' are computed as follows:

$$\begin{array}{l}\mathsf{broken}(F = V,\ T_s,\ T) \leftarrow \\ \quad \mathsf{terminatedAt}(F = V,\ T_f),\ \ T_s < T_f \leq T\end{array} \tag{1}$$

$$\begin{array}{l}\mathsf{broken}(F = V_1,\ T_s,\ T) \leftarrow \\ \quad \mathsf{initiatedAt}(F = V_2,\ T_f),\ \ T_s < T_f \leq T,\ \ V_1 \neq V_2\end{array} \tag{2}$$

$\mathsf{broken}(F = V, T_s, T)$ represents that $F = V$ is terminated at some time T_f such that $T_s < T_f \leq T$. Similar to initiatedAt, terminatedAt rules are domain-specific (examples are presented below). According to rule (2), if $F = V_2$ is initiated at T_f then effectively $F = V_1$ is terminated at time T_f, for all other possible values V_1 of F. Thus, rule (2) ensures that a fluent cannot have more than one value at any time. We do not insist that a fluent must have a value at every time-point. There is a difference between initiating a Boolean fluent $F = \mathsf{false}$ and terminating $F = \mathsf{true}$: the former implies, but is not implied by, the latter.

In what follows, we illustrate the use of RTEC for CE representation in the maritime domain.

Scenario 1. In maritime surveillance, it is necessary to detect areas in which vessel activity is suspicious. Below is the formalization of one type of suspicious activity:

$$\begin{array}{l}\mathsf{initiatedAt}(suspicious(Area) = \mathsf{true},\ T) \leftarrow \\ \quad \mathsf{happensAt}(\mathsf{start}(stopped(Vessel) = \mathsf{true}),\ T), \\ \quad \mathsf{holdsAt}(coord(Vessel) = (Lon, Lat),\ T), \\ \quad close(Lon, Lat, Area), \\ \quad \mathsf{holdsAt}(vesselsStoppedIn(Area) = N,\ T),\ N > 3 \\ \mathsf{terminatedAt}(suspicious(Area) = \mathsf{true},\ T) \leftarrow \\ \quad \mathsf{happensAt}(\mathsf{end}(stopped(Vessel) = \mathsf{true}),\ T), \\ \quad \mathsf{holdsAt}(coord(Vessel) = (Lon, Lat),\ T), \\ \quad close(Lon, Lat, Area), \\ \quad \mathsf{holdsAt}(vesselsStoppedIn(Area) = N,\ T),\ N \leq 3\end{array} \tag{3}$$

$stopped(Vessel) = \mathsf{true}$ is a durative ME stating that $Vessel$ has stopped. $\mathsf{start}(F = V)$ (respectively $\mathsf{end}(F = V)$) is a built-in RTEC event taking place at each starting (ending) point of each maximal interval for which $F = V$ holds continuously. Along with each

Table 1: Main predicates of RTEC.

Predicate	Meaning
$\mathsf{happensAt}(E,\ T)$	Event E occurs at time T
$\mathsf{holdsAt}(F = V,\ T)$	The value of fluent F is V at time T
$\mathsf{holdsFor}(F = V,\ I)$	I is the list of the maximal intervals for which $F = V$ holds continuously
$\mathsf{initiatedAt}(F = V,\ T)$	At time T a period of time for which $F = V$ is initiated
$\mathsf{terminatedAt}(F = V,\ T)$	At time T a period of time for which $F = V$ is terminated

vessel critical ME, the trajectory event detection system provides the coordinates (Lon, Lat) of the vessel. These are represented in RTEC by the $coord$ fluent. $close(Lon, Lat, Area)$ is an atemporal predicate calculating whether the Haversine distance between a point (Lon, Lat) and an $Area$ is less than some predefined threshold. Fluent $vesselsStoppedIn(Area)$ records the number of vessels that have stopped in this $Area$ at some point in time.

According to rule-set (3), an $Area$ is said to be suspicious as long as at least four vessels have stopped close to, or in it. The value of four vessels was set by domain experts. Usually, officials monitoring vessel activity are familiar with potentially suspicious areas, such as areas where loitering takes place, and thus restrict[3] computation of the maximal intervals of the $suspicious$ fluent to these areas. The maximal intervals during which $suspicious(Area) = \mathsf{true}$ holds continuously are computed using the built-in RTEC predicate holdsFor from rule-set (3).

$\mathsf{initiatedAt}(F = V, T)$ does not necessarily imply that $F \neq V$ at T. Similarly, $\mathsf{terminatedAt}(F = V, T)$ does not necessarily imply that $F = V$ at T. Suppose that $F = V$ is initiated at time-points 10 and 20 and terminated at time-points 25 and 30 (and at no other time-points). In that case $F = V$ holds at all T such that $10 < T \leq 25$. Note that, in this case, the event $\mathsf{start}(F = V)$ takes place at 10 and at no other time-point, while the event $\mathsf{end}(F = V)$ takes place at 25 and at no other time-point.

CE recognition for maritime surveillance requires reasoning over streaming data, such as the MEs reported by the trajectory event detection system, as well as atemporal reasoning [14]. In rule-set (3), for instance, we had to compute the Haversine distance between a point and an area. Unlike various other CE recognition approaches, such as [12, 18, 9], which lack the ability of (complex) reasoning over static/domain knowledge, RTEC combines event pattern matching over event streams with atemporal reasoning.

Scenario 2. Marine authorities are often interested in detecting illegal fishing. Below is a formalization of two of the conditions in which illegal fishing starts being recognized:

$$\begin{array}{l}\mathsf{initiatedAt}(illegalFishing(Area) = \mathsf{true},\ T) \leftarrow \\ \quad \mathsf{happensAt}(\mathsf{start}(stopped(Vessel) = \mathsf{true}),\ T), \\ \quad fishing(Vessel), \\ \quad \mathsf{holdsAt}(coord(Vessel) = (Lon, Lat),\ T), \\ \quad close(Lon, Lat, Area) \\ \mathsf{initiatedAt}(illegalFishing(Area) = \mathsf{true},\ T) \leftarrow \\ \quad \mathsf{happensAt}(slowMotion(Vessel),\ T), \\ \quad fishing(Vessel), \\ \quad \mathsf{holdsAt}(coord(Vessel) = (Lon, Lat),\ T), \\ \quad close(Lon, Lat, Area)\end{array} \tag{4}$$

[3]Such a restriction is achieved through the 'declarations' facility of RTEC.

fishing is an atemporal predicate indicating fishing vessels. In addition to having a database of *fishing* facts, this predicate may compute whether a vessel (not recorded in the database) is a fishing one given its characteristics. $slowMotion(Vessel)$ is an ME indicating that $Vessel$ was moving 'too' slowly at some point in time (see Section 3). The computation of the maximal intervals during which $illegalFishing(Area) = \mathsf{true}$ holds continuously is restricted to areas in which fishing is forbidden. According to rule-set (4), illegal fishing starts being recognized when a fishing vessel stops or moves 'too' slowly close to, or in an area in which fishing is forbidden.

Illegal fishing stops being recognized when there are no fishing vessels in the forbidden fishing area, or when their movement does not allow for fishing. The termination rules are formalized similar to the rules already shown and are therefore omitted to save space.

Scenario 3. A common feature of illegal shipping is communication gap; vessels with illegal activity, such as those passing through protected areas in order to minimize the length of a trip and therefore fuel consumption, switch off their transmitters and stop sending position signals. In such cases, it is often claimed that the transmitter temporarily broke down. To capture this type of activity, we defined the rule below:

$$\begin{aligned}
&\mathsf{happensAt}(illegalShipping(Area),\ T) \leftarrow \\
&\quad \mathsf{happensAt}(gap(Vessel),\ T), \\
&\quad \mathsf{holdsAt}(coord(Vessel) = (Lon, Lat),\ T), \\
&\quad close(Lon, Lat, Area)
\end{aligned} \tag{5}$$

$gap(Vessel)$ is an ME—the trajectory detection component reports such an ME when the $Vessel$ stops sending signals, i.e. when the communication gap starts. The computation of the time-points of the $illegalShipping(Area)$ events is restricted to protected areas, such as the National Marine Park of Alonnisos in the Aegean sea. According to rule (5), $illegalShipping(Area)$ is recognized when a vessel stops reporting position signals close to a protected $Area$.

Scenario 4. Ships sometimes approach inadvertently or intentionally (to reduce the length of a trip and fuel consumption) 'too' shallow waters. To alert marine authorities and prevent accidents resulting from such type of shipping, we formalized the rule below:

$$\begin{aligned}
&\mathsf{happensAt}(dangerousShipping(Area),\ T) \leftarrow \\
&\quad \mathsf{happensAt}(slowMotion(Vessel),\ T), \\
&\quad shallow(Area, Vessel), \\
&\quad \mathsf{holdsAt}(coord(Vessel) = (Lon, Lat),\ T), \\
&\quad close(Lon, Lat, Area)
\end{aligned} \tag{6}$$

$shallow$ is an atemporal predicate indicating whether some waters are 'too' shallow for a vessel. Similar to the *fishing* predicate, in addition to having a database of $shallow$ facts, we may compute whether an area is 'too' shallow for a vessel (not recorded in the database) given the vessel's characteristics. In rule (6), we chose the $slowMotion$ ME as a condition for the recognition of dangerous shipping. Similarly, we may add rules with other types of vessel movement for recognizing this CE.

4.2 Recognizing Maritime Activities

CE recognition may be performed retrospectively—e.g., at the end of each day in order to evaluate the activity of a particular fleet of vessels. Typically, though, CE recognition has to be efficient enough to support real-time decision-making, and scale to very large numbers of MEs and CEs. MEs may not necessarily arrive at the CE recognition system in a timely manner, i.e. there may be a (variable) delay between the time at which MEs take place and the time at which they arrive at the CE recognition system.

RTEC performs CE recognition by computing and storing the maximal intervals of fluents and the time-points in which events occur. CE recognition takes place at specified query times $Q_1, Q_2, \ldots$. At each Q_i the MEs that fall within a specified sliding window ω ("working memory" in the terminology of RTEC) are taken into consideration. All MEs that took place before or at $Q_i-\omega$ are discarded. This is to make the cost of CE recognition dependent only on range ω and not on the complete ME history.

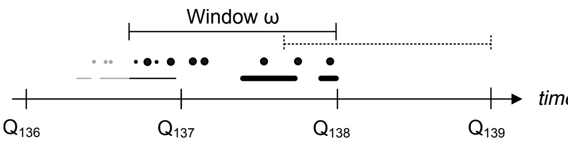


Figure 5: Complex event recognition in RTEC.

Window parameters for range ω and slide step β are specified by the users along with the definition of events, since they actually reflect the time horizon over which interesting phenomena shall be detected. Usually, range ω could span many minutes or even several hours for capturing meaningful events across a vessel's route.

At Q_i, the maximal intervals computed by RTEC are those that can be derived from MEs that occurred in the interval $(Q_i-\omega, Q_i]$, as recorded at time Q_i. When the range ω is longer than the slide step β, it is possible that an ME occurs in the interval $(Q_i-\omega, Q_{i-1}]$ but arrives at RTEC only after Q_{i-1}; its effects are taken into account at query time Q_i. This is illustrated in Figure 5. Occurrences of MEs are displayed as dots and a Boolean fluent as line segments. For CE recognition at Q_{138}, only the events marked in black are considered, whereas the greyed out events are neglected. Assume that all events marked in bold arrived only after Q_{137}. Then, we observe that two MEs were delayed, i.e., they occurred before Q_{137}, but arrived only after Q_{137}. In our setting, the window range ω is larger than the slide step. Hence, these events are not lost but considered as part of the recognition process at Q_{138}.

In the common case that MEs arrive at RTEC with delays, it is thus preferable to make the range of ω longer than the slide step. Note that information may still be lost. Any MEs arriving between Q_{i-1} and Q_i are discarded at Q_i if they took place before or at $Q_i-\omega$. To reduce the possibility of losing information, one may increase the window range ω. But doing so, decreases recognition efficiency. This issue is illustrated in the following section.

5. EMPIRICAL EVALUATION

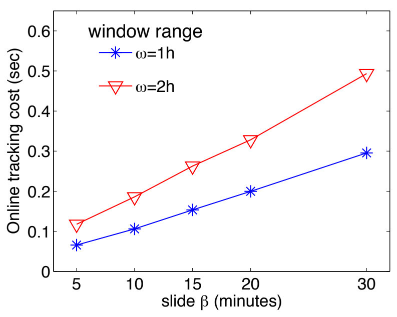
Our maritime surveillance system has a modular design with loosely coupled components. The online component for trajectory detection is developed in GNU C++ and runs entirely on main memory for efficiently coping with massive, volatile, streaming locations. RTEC, the CE recognition component, is implemented in YAP Prolog[4]. Built on top of PostgreSQL, Hermes MOD[5] accepts feeds of critical points from a Java wrapper and performs offline processing through SQL queries and stored procedures.

We conducted experiments against a real AIS dataset obtained from IMIS Hellas[6], our partner in the AMINESS project. Raw data is 23GB in size and spans from 1 June 2009 to 31 August 2009 for $N = 6425$ vessels in the Aegean, the Ionian, and part of the Mediterranean Sea. Not all vessels were actually on the move at all times, since a considerable part (chiefly cargo ships) were just passing by, and thus tracked for a limited period (days or even hours).

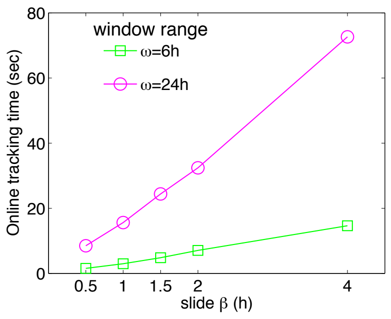
[4]The source code of RTEC, along with several sample CE definitions, is available at `http://users.iit.demokritos.gr/~a.artikis/EC.html`.

[5]Available at `https://hermes-mod.java.net/`

[6]`http://www.imishellas.gr/`



(a) Small window ranges



(b) Large window ranges

Figure 6: Online mobility tracking cost per window.

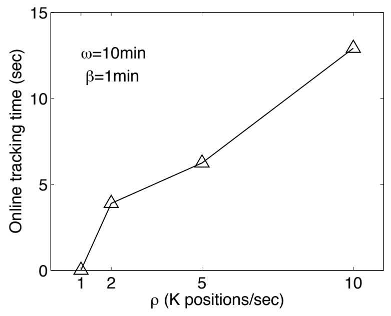


Figure 7: Varying arrival rates.

Table 2: Experimental settings.

Parameter	**Value**
Vessel count N	6425
Window range ω	10min, 1h, 2h, **6h**, 9h, 24h
Window slide β	1min, 5min, 10min, 15min, 20min, 30min, **1h**, 90min, 2h, 4h
Stream arrival rate ρ (positions/sec)	**original**, 1K, 2K, 5K, 10K

Table 3: Mobility tracking parameters.

Parameter	**Value**
Minimum speed v_{min} for asserting movement	1 knot (≅1.852 km/h)
Rate of speed change α	25%
Minimum gap period ΔT	10 minutes
Turn threshold $\Delta\theta$	5^o, 10^o, **15^o**, 20^o
Radius r for long-term stops	200 meters
Minimal number m of inspected positions	10

But most vessels were frequently sailing, e.g., passenger ships or ferries to the islands. When decoded and cleaned from corrupt messages, the dataset yielded 168,240,595 timestamped positions[7].

We simulated a streaming behavior by consuming this positional data little by little, i.e., reading small chunks periodically according to window specifications. We examine sliding windows with varying ranges ω and slide steps β based on timestamps from the original AIS messages. Thus, we replay this stream and the window keeps in pace with the reported timestamps and not the actual time of each simulation. The arrival rate of positions is fluctuating throughout this 3-month period and varies widely among vessels; none of them reports at a fixed frequency, whereas there are ships inactive for large intervals. On average, each vessel reports its position once every 2 minutes, by considering only its activity period (i.e., when it actually relays positions, either moving or not). This translates into a mean arrival rate ρ of about 50 positions/sec. For consistency with the real-world scenario, we consume the original stream "as is" in all our simulations, with the exception of one test conducted at artificially increased rates. Simulation settings are listed in Table 2, whereas calibrated settings for online mobility tracking are given in Table 3; default values are shown in bold.

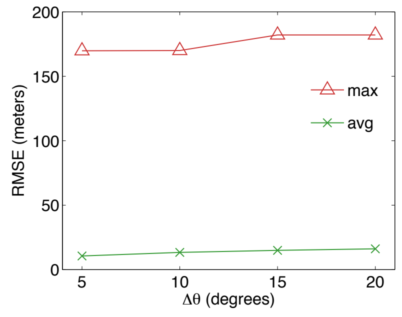
Next, we report indicative results from these experiments. The trajectory event detection component operated on a server running Debian Linux "Wheezy" 7.5 amd64 with two Intel Xeon X5675 processors at 3.07GHz. This machine has 48GB of RAM, and five hard disks at 15K RPM with RAID 0 and a total capacity of 3TB. The complex event recognition component operated on a computer with Intel i7-4770@3.40GHz×8 processors and 16GiB RAM, running Ubuntu Linux 14.04 and YAP Prolog 6.2.2.

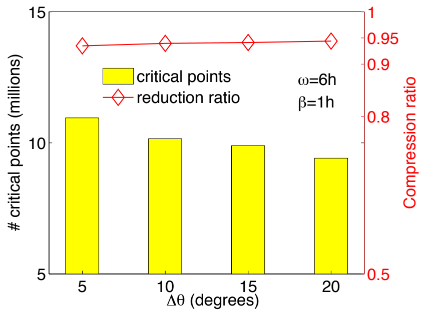
5.1 Assessment of Trajectory Detection

Performance of online tracking. The first set of experiments examines performance of online mobility tracking for window specifications with varying ranges ω and slide steps β. Figure 6 illustrates the execution cost for the entire fleet per window, i.e., how much time it takes to update the window with fresh locations, evict expired ones, detect trajectory events, and report critical points. All values are averages over the total count of window instantiations, so they represent the per slide cost for window maintenance and identification of any trajectory events therein. Simulations with the original arrival rate reveal that critical points are issued almost instantly for small ranges up to two hours (Figure 6(a)). Not surprisingly, this cost escalates linearly when the window slides forward less often (larger β), because the mobility tracker must check many more fresh positions spanning a wider period β. Yet, processing positional batches arrived over the past $\beta = 30$ minutes never takes more than 500ms for small window ranges. The same linear pattern in online tracking cost repeats with wider ranges (Figure 6(b)), but it takes more time to complete upon each slide. In the worst case of a window spanning 24 hours, critical points are reported in only 72 seconds based on the bulk of data accumulated over each 4-hour period, which clearly demonstrates the robustness of this method.

One might argue that such performance results should be expected, given the low rate of the original stream. For a more stringent assessment of the online mobility tracking module, we performed an extra simulation, by admitting bigger chunks of data for processing at considerably increased arrival rates up to ρ =10,000 positions/sec. Given the fleet size N, every ship appears as reporting almost twice per second. This is quite improbable in practice, but makes sense as a stress test. As our objective is timeliness, the window was set to span $\omega = 10$ minutes and to slide each minute. In Figure 7, observe that critical points are still issued promptly for $\rho = 1,000$ positions/sec, but the latency grows with increasing rates. Reporting cost for critical points (i.e., cost after detection) is included in these times, and this becomes a significant overhead when massive AIS updates inevitably generate more critical points. For $\rho = 10,000$ positions/sec, the online tracker has to deal with 600,000 fresh positions every $\beta = 1$ minute, which is undoubtedly a demanding task. Nonetheless, it never takes more than a few seconds to respond, well before the next window slide. This behavior confirms that the trajectory detection process is capable of handling scalable volumes of streaming vessel positions.

[7]This anonymized data (with MMSIs replaced by sequence numbers) is publicly available at `http://www.chorochronos.org/?q=content/imis-3months`





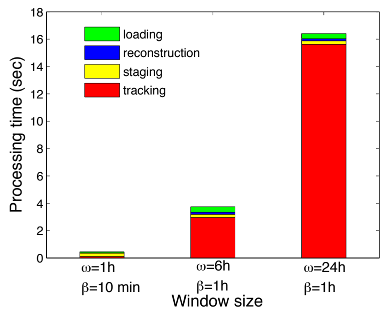


Figure 8: Trajectory approximation error. **Figure 9: Compression for varying $\Delta\theta$.** **Figure 10: Trajectory maintenance cost.**

Approximation error. Apart from performance, we also assessed the quality of trajectories approximately reconstructed from critical points only. For the entire motion history of each vessel, we estimated deviation between the original trajectory and its approximate representation (i.e., after compression). Deviation between two polylines can be computed from the pairwise distance of their corresponding points; in our case, between each pair of *synchronized* locations. Suppose that an original AIS point p_i did not qualify as critical and was discarded at timestamp τ_i. To estimate the resulting deviation for a particular vessel, we interpolated between the pair of adjacent critical points retained immediately before and after each such p_i. Assuming a constant velocity between these two critical points, we obtained its time-aligned point trace p'_i along the approximate path at timestamp τ_i. Thus, the compressed trajectory retained its approximate shape, but with additional (synchronized) vertices that account for the evicted positions. Assuming that a vessel originally reported M positions, we estimated the root mean square error ($RMSE$) between original and synchronized sequences of its locations using this formula:

$$RMSE = \sqrt{\frac{1}{M} \cdot \sum_{i=1}^{M} (H(p_i, p'_i))^2}$$

where H denotes the Haversine distance between geographic coordinates, and returns RMSE estimates in meters. One error value was computed per vessel trajectory; Figure 8 plots the *average* and *maximum* $RMSE$ of them for several values of turn threshold $\Delta\theta$, which is used to recognize significant changes in heading. As discussed in Section 3.1, the degree of trajectory approximation is sensitive mostly to angle $\Delta\theta$ compared to other parameters in Table 3. In our tests, average error along complete trajectories (i.e., the entire motion history of each vessel) never exceeds 16 meters. This is negligible compared to the much larger size of most ships, and also considering the discrepancies inherent in GPS positioning. The maximum $RMSE$ ever observed is 182 meters, under a relaxed sensitivity of $\Delta\theta = 20^o$ for capturing important turns only. Although such a threshold is rather wide for actual monitoring of maritime activity, the worst error among all trajectories is comparable to the length of large ships. So, it turns out that the online tracking component provides quite acceptable accuracy and can capture most, if not all, critical changes along each vessel's course.

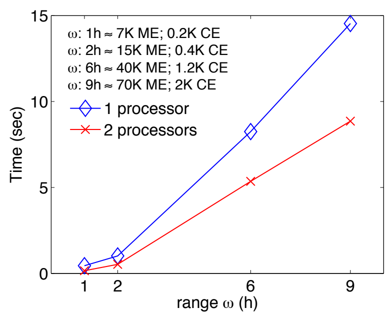
Compression efficiency. In this experiment, we examine the efficiency of our prototype in keeping only major trajectory characteristics as critical points and discard the rest. In order to measure the *compression ratio* accomplished by online trajectory tracking, we compared the amount of discarded points against the originally relayed locations per vessel. A compression ratio close to 1 signifies stronger data reduction, as the vast majority of original locations are dropped. The line plot in Figure 9 depicts measurements of this ratio with varying tolerance angles for detecting heading changes. With a lower $\Delta\theta$, even slight deviations in vessel direction can be spotted, and thus extra critical points get reported. From the bar plot in Figure 9, we notice that every further increase by 5^o in turn threshold $\Delta\theta$ results in about 5% drop in the total amount of critical points. So, relaxing the parameter values leads to a slightly less intense compression. However, compression ratio remains close to 94%, which means that about 6% of the original locations only survive as critical. In a streaming context, such high compression may lead to reduced system load in subsequent stages of the analysis, and we stress that it comes without significant loss in quality, as discussed earlier.

Table 4: Statistics from compressed trajectories.

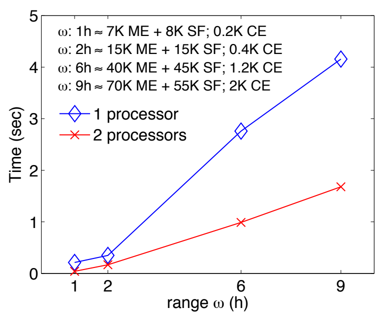
Critical points in reconstructed trajectories	7,776,947
Critical points remaining in staging area	2,524,925
Number of trips between ports	68,501
Average trips per vessel	30
Average number of critical points per trip	113
Average travel time per trip	1 day 07:20:58
Average traveled distance per trip	221.976km

Trajectory maintenance. Since the system accepts streaming positions from vessels and manages to maintain their historical trajectories, we now provide some evidence of its overhead. Figure 10 plots the average processing cost per window slide for all four phases. Evidently, online mobility *tracking* undertakes the hardest task by filtering the huge volume of incoming positions, hence it dominates the trajectory maintenance cost especially for greater window sizes. As argued before, tracking time escalates with the window range, and also increases linearly when the window slides less often. A batch of "delta" critical points evicted from the sliding window is transferred into the *staging* area (on disk) in less than 260ms. This cost does not fluctuate a lot, as it mainly depends on database connection tunnelling via the Java wrapper. Trajectory *reconstruction* into trips between ports is also quite efficient and takes at most 163ms per batch of critical points. This offline module has to consider a drastically reduced amount of critical points per vessel instead of the voluminous dataset of raw positions, hence it incurs little overhead. In the last stage of *loading*, trajectory segments are inserted or updated in Hermes MOD, and this is also fast (390ms in the worst case), thanks to the reduced data volumes involved.

In Table 4, we list representative statistics from trajectories reconstructed and archived in the database. This computation took place after the input stream was exhausted and all critical points



(a) Input: critical movement events (ME).



(b) Input: critical movement events (ME) and spatial facts (SF).

Figure 11: Complex event recognition for 6,425 vessels and 35 areas.

were detected for the entire 3-month period. An outstanding benefit is that, with only a moderate amount of points, we can approximately describe trips spanning more than a day and covering long distances. Note that about 25% of critical points have not been assigned into any trajectory. These come from vessels still sailing and having not yet reached their destination port ("open-ended" trips). Besides, the number of trips is an order of magnitude greater than fleet size N. This certainly increases the amount of records that must be stored in the trajectory database, but thanks to their semantic enrichment with port information, such shorter geometry representations are more preferable at query time than protracted sequences of featureless locations.

5.2 Complex Event Recognition Performance

The task of the complex event (CE) recognition module is to detect activities of special significance for maritime surveillance, given the critical movement events (ME) produced by the trajectory event detection component. The input of RTEC—our CE recognition module—consists of the MEs (communication) *gap*, *lowSpeed*, *stopped*, *speedChange* and *turn*, as well as the coordinates of each vessel at the time of ME detection. Given such an event stream, RTEC recognizes suspicious vessel activity (several vessels stopped in some area), illegal fishing, illegal shipping (passing through protected areas) and dangerous shipping (sailing through shallow waters). The choice of CEs and their definitions (see Section 4) were specified in collaboration with the domain experts of the AMINESS project. To allow for the recognition of the aforementioned CEs, we enhanced the input of RTEC with static synthetic data. For each vessel we added information about its draft, while a number of vessels were designated as fishing vessels. Moreover, we generated 35 polygons representing protected areas, forbidden fishing areas, and areas with shallow waters.

Figure 11(a) shows the results of two sets of experiments. First, we used a single processor to perform CE recognition for all 6,425 vessels and 35 areas. Second, we used two processors of the computer on which RTEC operated in parallel. One processor performed CE recognition for the areas located in, and the vessels passing through the west part of the area under surveillance. Similarly, the other processor performed CE recognition for the areas located in, and the vessels passing through the east part of the monitored area. Figure 11(a) shows average CE recognition times in CPU seconds. The slide β is 1 hour, including approximately 7,000 MEs. The window range ω varies from 1 hour ($\approx$ 7,000 MEs) to 9 hours ($\approx$ 70,000 MEs). In the distributed setting—two processors for CE recognition—the input MEs are forwarded to the appropriate processor (according to vessel location). The number of CEs also depends on the window range. For $\omega = 1$ hour, approximately 200 CEs are recognized, while for $\omega = 9$ hours RTEC recognizes approximately 2,000 CEs.

Figure 11(a) shows that we can achieve a significant performance gain by running RTEC in parallel. The input to each processor is restricted to the MEs of the vessels for which it performs CE recognition. Furthermore, each processor has to compute and store the maximal intervals of a smaller number of CEs. One may further distribute CE recognition by dividing further the monitored area, thus reducing CE recognition times. Figure 11(a) also shows that RTEC is capable of supporting real-time CE recognition. E.g., for a window ω of 6 hours, RTEC recognizes all CEs requested by end users in 8 sec when a single processor is used, and in 5 sec when two processors are used in parallel.

Note that, since the input stream consists of *critical* MEs, most of them fire the CE definition rules. This is in contrast to other experiments [5, 6] where the input stream includes several events that do not affect CE recognition. CE recognition performance does not depend entirely on the size of the input data streams. The complexity of the CE definitions affects recognition times significantly. In this work, we make use of quite complex CE definitions including various constraints on vessels and areas. This is in contrast to the majority of the event processing literature where quite simple CE definitions are used for empirical analysis.

Figure 11(b) shows average CE recognition times without spatial reasoning. More precisely, the ME stream is augmented by timestamped facts indicating the spatial relations between vessels and (protected, forbidden fishing, shallow) areas. Each ME expressing the movement of a vessel is accompanied by facts stating whether the vessel is 'close' to some area of interest—the timestamp of these facts is the same as the timestamp of the ME. For these experiments, the CE definitions were updated in order to make use of spatial facts (as opposed to RTEC computing on-demand spatial relations in the CE recognition process). Figure 11(b) shows results concerning CE recognition by single processor, as well as CE recognition performed by two processors in parallel. The slide β is 1 hour. In this setting, however, 1 hour of data includes approximately 15,000 input facts: 7,000 MEs and 8,000 spatial facts. Moreover, the window range ω varies from 15,000 MEs and spatial facts (1 hour) to 125,000 MEs and spatial facts (9 hours). As expected, the number of recognized CEs does not change with respect to the experiments including spatial reasoning.

Figure 11(b) shows that even though the stream used as input for CE recognition increases significantly when RTEC does not perform spatial reasoning, the average CE recognition times decrease substantially. Moreover, this figure shows that RTEC scales to large data streams—e.g. CE recognition for all 6,425 vessels and 35 areas using a window of 125,000 input facts takes on average 1.5 sec when two processors are used in parallel.

5.3 Discussion

Empirical results confirm that the proposed system is capable of recognizing, in real-time, suspicious and potentially dangerous situations that require immediate attention from marine authorities. Even when it must cope with an increased amount of incoming positions (expressed with larger window ranges sliding less often), it can recognize complex events in a few seconds at most. Such performance is noteworthy, given the large number of monitored vessels in the simulated scenario. This comes thanks to the scalability and robustness of the trajectory event detection mechanism, which offers reliable incremental response (often in less than a second) and can filter out noisy or intermittent input signals. In addition, the complex event recognition module can take advantage of the evolving trajectory features in order to recognize complex spatiotemporal relationships between vessels and areas of interest.

6. RELATED WORK

Detecting events from massive, streaming data has attracted a lot of research interest, but also opens up great perspectives for building powerful monitoring applications in several domains. Event detection from both live and archived streams has been proposed in [10], introducing optimizations specifically for recency-probing pattern queries. The goal of UpStream platform [23] is to answer continuous queries with the lowest staleness possible, when each data item represents an update to a previous one; this certainly applies to GPS positions from moving vessels, but UpStream lacks support for the specific demands of trajectory monitoring. Automating ingestion of streaming data feeds from various sources into data warehouses is also a challenging issue, as outlined in a recent tutorial [15]. To the best of our knowledge, no streaming framework is specifically tailored for maritime surveillance over fluctuating, noisy, intermittent AIS messages from large fleets.

Detecting trajectory events from positional streams essentially performs path simplification, a topic explored in several previous works. Some strategies opt for acceptable approximations in terms of a given error margin, such as those in [8, 19, 22]. Besides, the memory space available for retaining a compressed sequence may also be crucial in a single-pass evaluation [31]. Dead-reckoning policies like [33] are employed in moving sources to relay positional updates upon significant deviation from the course already known to a centralized server, so they aim to reduce communication cost. This is not the case with AIS data, as maritime control centers wish to locate ships as frequently as possible. The advantage of our proposed scheme is that it accounts for stream imperfections, i.e., the noise inherent in vessel positions due to sea drift, delayed arrival of messages, or discrepancies in GPS signals. Most importantly, we annotate reduced representations according to particular movement events along each vessel trace.

For archiving trajectories, we make use of Hermes [30], a prototype Moving Object Database (MOD) equipped with a powerful query language. Hermes MOD supports modeling and querying of moving objects, and enables support of aggregative Location-Based Services. It defines a trajectory data type as well as a collection of spatiotemporal operations (range, nearest neighbor, similarity, etc.), which take advantage of a robust indexing mechanism. Semantic-aware trajectory construction [34] applies cleaning, compression and segmentation over positional data, in order to define "stop" and "move" episodes along each trace in online fashion. Besides, a formal model for OLAP operations at different granularities was recently proposed in [20], but it is mostly geared towards visual analytics over offline trajectory data.

In terms of Complex Event Recognition, RTEC has a formal, declarative semantics in contrast to other related systems that usually rely on an informal and/or procedural semantics. Cugola and Margara [9] point out that almost all "complex event processing languages", including [4], and several "data stream processing languages", such as ESL [7], lack a rigorous, formal semantics. Eckert and Bry [13] note that the semantics of "event query languages" often are somewhat ad hoc, unintuitive and generally have an algebraic and less declarative flavor. Paschke and Kozlenkov [27] state that commercial "production rule languages" lack a declarative semantics. Unlike [12, 18, 9], RTEC supports complex atemporal reasoning and reasoning over background knowledge (e.g., identifying the type of a vessel given its characteristics), which are quintessential for maritime surveillance [14]. This way, it is possible to express the complex phenomena required by the maritime experts [32]. Furthermore, RTEC explicitly represents complex event intervals and thus avoids the related logical problems (see [25] for a discussion of these problems), and supports out-of-order event streams (in contrast to e.g. [11, 9, 10, 21]). Concerning the Event Calculus literature, a key feature of RTEC is that it includes a windowing semantics. In contrast, no Event Calculus system "forgets" or represents concisely the event history.

7. SUMMARY & FUTURE WORK

In this paper, we introduced a system that monitors the activity of thousands of vessels and can instantly detect and recognize events with a potentially serious impact on the environment and on safe navigation at sea. The system can sustain large amounts of streaming messages from vessels and can filter out noise and redundant positions along their course. Hence, it can retain only succinct synopses of vessel trajectories, drastically reducing the original path into few critical points that convey major motion characteristics. Furthermore, this reduced information may be readily analyzed online for complex event recognition. Equipped with efficient pattern matching algorithms, this module correlates critical trajectory positions with static geographical and vessel data, and detects suspicious or dangerous situations, such as loitering, vessels passing through protected areas, and unsafe shipping. Our platform has been empirically validated against a large real dataset and met expectations for timeliness, scalability, and robustness.

We plan further extensions and improvements in the existing implementation. First, we soon expect to be given access to *live* AIS feeds from all vessels across the Aegean Sea. This will integrate our system with a precious source of online data, offering to marine experts and authorities the means to instantly locate, recognize, and correlate events from real-time vessel traces.

Existing definitions of complex events were manually developed in collaboration with vessel traffic service staff. However, creating CE definitions manually is painstaking and error-prone. We have begun exploring techniques based on abductive-inductive logic programming, for automated generation and refinement of definitions from very large datasets.

Besides, maritime surveillance exhibits various types of uncertainty, exactly like most event processing applications. So, we are porting RTEC into probabilistic logic programming frameworks, in order to deal with imperfect complex event definitions, incomplete and erroneous data streams. A probabilistic treatment would be also challenging for addressing the gradual ageing and transmission delays in AIS data. Traffic forecasts at short-term horizons (e.g., 5, 15, or 30 minutes ahead) could also be issued, gracefully weighing online events with offline trajectory analytics.

Last, but not least, maritime surveillance may benefit from combining multiple data sources. As RTEC readily supports heterogeneous stream processing [6], we aim to experiment with additional data, such as weather forecasts, for improved monitoring.

Acknowledgements

This work was carried out in the framework of the project *"AMINESS: Analysis of Marine INformation for Environmentally Safe Shipping"* co-financed by the European Fund for Regional Development and from Greek National funds through the operational programs "Competitiveness and Enterpreneurship" and "Regions in Transition" of the National Strategic Reference Framework - Action: "COOPERATION 2011 – Partnerships of Production and Research Institutions in Focused Research and Technology Sectors". An initial version of the online tracking module was partially developed during a Short Term Scientific Mission (STSM) of the first author at Naval Academy Research Institute (Brest, France) with the support of EU COST Action IC0903 on Knowledge Discovery from Moving Objects (MOVE). Helpful discussions with Cyril Ray and Christophe Claramunt in this STSM are kindly acknowledged.

8. REFERENCES

[1] Aminess project. `http://www.aminess.eu/`.

[2] Marine traffic. `http://www.marinetraffic.com/`.

[3] Seabilla project. `http://www.seabilla.eu/`.

[4] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *SIGMOD*, pages 147–160, 2008.

[5] A. Artikis, M. Sergot, and G. Paliouras. An event calculus for event recognition. *IEEE Transactions on Knowledge and Data Engineering*, 2014.

[6] A. Artikis, M. Weidlich, F. Schnitzler, I. Boutsis, T. Liebig, N. Piatkowski, C. Bockermann, K. Morik, V. Kalogeraki, J. Marecek, A. Gal, S. Mannor, D. Gunopulos, and D. Kinane. Heterogeneous stream processing and crowdsourcing for urban traffic management. In *EDBT*, pages 712–723, 2014.

[7] Y. Bai, H. Thakkar, H. Wang, C. Luo, and C. Zaniolo. A data stream language and system designed for power and extensibility. In *CIKM*, pages 337–346, 2006.

[8] H. Cao, O. Wolfson, and G. Trajcevski. Spatio-temporal data reduction with deterministic error bounds. *VLDB Journal*, 15(3):211–228, 2006.

[9] G. Cugola and A. Margara. TESLA: a formally defined event specification language. In *DEBS*, pages 50–61, 2010.

[10] N. Dindar, P. M. Fischer, M. Soner, and N. Tatbul. Efficiently correlating complex events over live and archived data streams. In *DEBS*, pages 243–254, 2011.

[11] L. Ding, S. Chen, E. A. Rundensteiner, J. Tatemura, W.-P. Hsiung, K. Candan. Runtime semantic query optimization for event stream processing. In *ICDE*, pages 676–685, 2008.

[12] C. Dousson and P. Le Maigat. Chronicle recognition improvement using temporal focusing and hierarchisation. In *IJCAI*, pages 324–329, 2007.

[13] M. Eckert and F. Bry. Rule-based composite event queries: the language xchange^{eq} and its semantics. *Knowledge Information Systems*, 25(3):551–573, 2010.

[14] J. Garcia, J. Gomez-Romero, M.A. Patricio, J.M. Molina, and G. Rogova. On the representation and exploitation of context knowledge in a harbor surveillance scenario. In *FUSION*, pages 1–8, 2011.

[15] L. Golab and T. Johnson. Data stream warehousing *(tutorial)*. In *ACM SIGMOD*, pages 949–952, 2013.

[16] B. Idiri and A. Napoli. The automatic identification system of maritime accident risk using rule-based reasoning. In *SoSE*, pages 125–130, 2012.

[17] R. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4(1):67–96, 1986.

[18] J. Krämer and B. Seeger. Semantics and implementation of continuous sliding window queries over data streams. *ACM Transactions on Database Systems*, 34(1):1–49, 2009.

[19] R. Lange, F. Dürr, and K. Rothermel. Efficient real-time trajectory tracking. *VLDB Journal*, 20(5):671–694, 2011.

[20] L. Leonardi, S. Orlando, A. Raffaetà, A. Roncato, C. Silvestri, G.L. Andrienko, and N.V. Andrienko. A general framework for trajectory data warehousing and visual OLAP. *GeoInformatica*, 18(2):273–312, 2014.

[21] M. Li, M. Mani, E. A. Rundensteiner, and T. Lin. Complex event pattern detection over streams with interval-based temporal semantics. In *DEBS*, pages 291–302, 2011.

[22] N. Meratnia and R.A. de By. Spatiotemporal compression techniques for moving point objects. In *EDBT*, pages 765–782, 2004.

[23] A. Moga and N. Tatbul. UpStream: A storage-centric load management system for real-time update streams. *PVLDB*, 4(12):1442–1445, 2011.

[24] International Maritime Organization. Automatic identification systems. `http://www.imo.org/OurWork/Safety/Navigation/Pages/AIS.aspx`.

[25] A. Paschke. ECA-RuleML: An approach combining ECA rules with temporal interval-based KR event/action logics and transactional update logics. Technical Report 11, TU München, 2005.

[26] A. Paschke and M. Bichler. Knowledge representation concepts for automated SLA management. *Decision Support Systems*, 46(1):187–205, 2008.

[27] A. Paschke and A. Kozlenkov. Rule-based event processing and reaction rules. In *RuleML*, pages 53–66, 2009.

[28] K. Patroumpas. Online tracking and summarization over streaming maritime trajectories. In *MOVE Workshop on Moving Objects at Sea*, 2013.

[29] K. Patroumpas and T. Sellis. Maintaining consistent results of continuous queries under diverse window specifications. *Information Systems*, 36(1):42–61, 2011.

[30] N. Pelekis, E. Frentzos, N. Giatrakos, and Y. Theodoridis. Hermes: Aggregative LBS via a trajectory DB engine. In *ACM SIGMOD*, pages 1255–1258, 2008.

[31] M. Potamias, K. Patroumpas, and T. Sellis. Online amnesic summarization of streaming locations. In *SSTD*, pages 148–165, 2007.

[32] J. van Laere and M. Nilsson. Evaluation of a workshop to capture knowledge from subject matter experts in maritime surveillance. In *FUSION*, pages 171–178, 2009.

[33] O. Wolfson, A.P. Sistla, S. Chamberlain, and Y. Yesha. Updating and querying databases that track mobile units. *Distributed & Parallel Databases*, 7(3):257–287, 1999.

[34] Z. Yan, N. Giatrakos, V. Katsikaros, N. Pelekis, and Y. Theodoridis. SeTraStream: Semantic-aware trajectory construction over streaming movement data. In *SSTD*, pages 367–385, 2011.

Identifying User Interests within the Data Space – a Case Study with SkyServer

Hoang Vu Nguyen[○] Klemens Böhm[○] Florian Becker[○]
Bertrand Goldman[◇] Georg Hinkel[○] Emmanuel Müller[○●]

[○] Karlsruhe Institute of Technology (KIT), Germany
{hoang.nguyen, klemens.boehm, georg.hinkel, emmanuel.mueller}@kit.edu and florian.becker@student.kit.edu

[◇] Max Planck Institute for Astronomy, Germany
goldman@mpia.de

[●] University of Antwerp, Belgium
emmanuel.mueller@ua.ac.be

ABSTRACT

Many scientific databases nowadays are publicly available for querying and advanced data analytics. One prominent example is the Sloan Digital Sky Survey (SDSS)—SkyServer, which offers data to astronomers, scientists, and the general public. For such data it is important to understand the public focus, and trending research directions on the subject described by the database, i.e., astronomy in the case of SkyServer. With a large user base, it is worthwhile to identify the areas of the data space that are of interest to users.

In this paper, we study the problem of extracting and analyzing *access areas* of user queries, by analyzing the query logs of the database. To our knowledge, both the concept of access areas and how to extract them have not been studied before. We address this by first proposing a novel notion of access area, which is independent of any specific database state. It allows the detection of interesting areas within the data space, regardless if they already exist in the database content. Second, we present a detailed mapping of our notion to different query types. Using our mapping on the SkyServer query log, we obtain a transformed data set. Third, we aggregate similar overlapping queries by DBSCAN and gain an abstraction from the raw query log. Finally, we arrive at clusters of access areas that are interesting from the perspective of an astronomer. These clusters occupy only a small fraction (in some cases less than 1%) of the data space and contain queries issued by many users. Some frequently accessed areas even do not exist in the space spanned by available objects.

1. INTRODUCTION

Nowadays, many scientific databases are made publicly available to reach a large community of users. Popular examples are SkyServer from astronomy and GBIF from biosystematics. With a large user base, it is of great benefit to identify the parts of the data space that many users are interested in. This data space is formed by the database schema, which defines the domain of each column and their relationships. Note that the data space is not constrained to the actual database content. For instance, Figure 1(a) plots the subspace formed by two columns *plate* and *mjd* of relation *SpecObjAll* of SkyServer. One can see that the database content does not span the whole data space, leaving an *empty area*, i.e., part of the data space containing no data object.

A good indicator of the interestingness of a (sub)area of the data space is the frequency it is referred to in user queries. Identifying common interests of many users is useful for traditional applications such as performance tuning and query personalization [4, 17, 22]. However, it also is important for learning the database usage, which tends to represent the focus of the respective scientific community. In fact, our study of the SkyServer query log reveals that the interests of users in the data space often correspond to a small part of the database content. Furthermore, users even are interested in *empty* areas of the data space. Inspecting data objects queries actually have retrieved does not give rise to such insights. The following example is an excerpt of our results.

EXAMPLE 1. *As shown in Figure 1(a), the content of relation* SpecObjAll *is within the area*

$$(266 \leq \mathit{SpecObjAll.mjd} \leq 5141) \\ \wedge (51578 \leq \mathit{SpecObjAll.plate} \leq 55752)$$

*of the subspace (*SpecObjAll.plate, SpecObjAll.mjd*) of the data space. An area returned by our algorithm that is accessed by* 18, 904 *queries is*

$$(296 \leq \mathit{SpecObjAll.plate} \leq 3200) \\ \wedge (51578 \leq \mathit{SpecObjAll.mjd} \leq 52178) \quad ,$$

i.e., a small part of the content of SpecObjAll.

*In Figure 1(b), the area of the subspace (*PhotoObjAll.ra, PhotoObjAll.dec*) that queries refer to spans not only its database content, but also its* empty area. *The number of such 'empty area' queries is significant, see Table 1 in the evaluation.*

Figure 1(c) depicts a scenario similar to that of Figure 1(b). However, here the empty areas of the subspace accessed by queries are not contiguous and are larger than the part occupied by the database content.

The access areas found can be used in various ways: They could help funding agencies to align the spending of their resources with community interests. In the context of SkyServer, priority could be given to projects exploring the area of the sky many astronomers/ scientists/students/the public are interested in. Further, they could help researchers to identify 'gaps' in the current scientific knowledge systematically and to select new, good research topics that are up to what many people are targeting.

The problem studied in this paper is to identify areas of interest to many users within the data space, at the right level of abstraction, from the query log. Our approach rests on the following pillars:

- The database offers a flexible query interface such as SQL. Such databases and means to query it flexibly are fundamentally important in many scientific domains.
- The database serves a large number of users, i.e., a large community. This ensures the value of the common interests extracted.
- Most users do not have personal contact with the database owner. Hence, he/she does not have an objective picture of what users really are after. In other words, the anecdotical evidence that he/she may have is not necessarily representative of the interests of the user group as a whole.
- Most users of a scientific database are knowledgeable on its domain, e.g., by working with domain experts with whom they have collaboration. This implies that most queries issued by users are meaningful. Furthermore, such databases limit the number of queries each user is allowed to issue within a window of time. This makes queries well-designed, but also hinders us to re-issue a large number of queries to collect statistics.

In such a setting, the problem we study can be more precisely phrased as: *Given the query log containing SQL statements issued by users, how to extract their intents, i.e., the areas of the data space many users wanted to access, without accessing (re-querying) the database?*

We see three main challenges in the way of solving this problem. First, we need to come up with a formal definition of access area, i.e., the area of the data space that a query refers to. The definition needs to be sufficiently abstract, so that it is not confined to a specific data model or even a certain database schema. Further, it must not be confined to the database content and enable the discovery of empty areas of the data space many users care about. Second, given an abstract definition of access area, we need to come up with a mapping of queries to their access areas, or, in other words, a realization of the definition on all query types actually occurring. This is far from obvious, due to the expressiveness of modern query languages. In fact, it turns out that in some cases the mapping is overly complex and sophisticated analysis is required. Third, we need a procedure to aggregate the access areas of a large set of queries. Clustering seems promising, but to do so, we need a distance measure. Such a distance measure needs to be defined, since existing ones for queries focus on their structure [4]. Here in turn, the focus has to be on the content (i.e., access areas) and the similarity of overlapping areas for a meaningful abstraction of a set of queries.

This paper represents our solution to each of these challenges. In particular, we introduce the novel concept of access area, which is independent of any specific data model and any database state. Being independent of the database content brings a performance gain compared to actually rerunning the queries. In addition, our notion of access area lets us achieve our important goal: We discover areas of the data space that are of interest to users, irrespective of the number of data objects falling into these areas. Second, we provide a mapping of our notion to all query types occurring in the SkyServer query log, i.e., we enable extraction of access areas in practice. We also show that extracting access areas is not straightforward for queries with joins, aggregate queries, and nested queries. Third, we exploit query overlap for the aggregation of access areas of a large set of queries.

At this point, our main interest is on the application side, i.e., to find out whether our approach makes sense for domain experts (it does), whether the results obtained so far are plausible (mostly yes), and whether they allow for new insights regarding user interests (not so much at this point, but there are promising starting points for refinements). We have learned this from a case study on SkyServer and interviewing both a person responsible for SkyServer and an 'independent' astronomer. In the study, we extract access areas from the SkyServer query log and cluster the transformed data with DBSCAN. The clustered access areas found occupy only a small fraction (in some cases less than 1%) of the data space and are accessed by many users. We also detect access areas which do not even contain any data objects. Such areas can be rather large (see Figure 1(c)). Finally, our astronomer does not only deem our approach helpful for the owner of the data, but also for users.

While our focus so far has been on SkyServer, our concepts are applicable to any database, i.e., no confinement to RDBMSs. Further, since no SkyServer-specific features have been hard-coded in our implementation, it is applicable to any SQL database and facilitates any future extension to cope with databases in other domains. We repeat that our concern with this paper is to propose an approach that leads to interesting results. Tuning the approach, e.g., by experimenting with other distance functions or clustering algorithms systematically, is beyond our current scope.

Paper outline: In Section 2, we provide our definition of acess area. In Section 3, we review related work. In Section 4, we describe our implementation. In Section 5, we present our distance function. Section 6 features our case study; Section 7 concludes.

2. ACCESS AREA

The access area of a query captures the area of the data space that the user is interested in. We now formalize this concept. To ease our presentation, we first introduce some notation. Then we discuss two straightforward ways to define access areas and point out their drawbacks. Finally, we propose our notion of access area.

2.1 Preliminaries

We consider a relational database $\mathbf{DB}$ which consists of multiple relations; each relation has several columns.

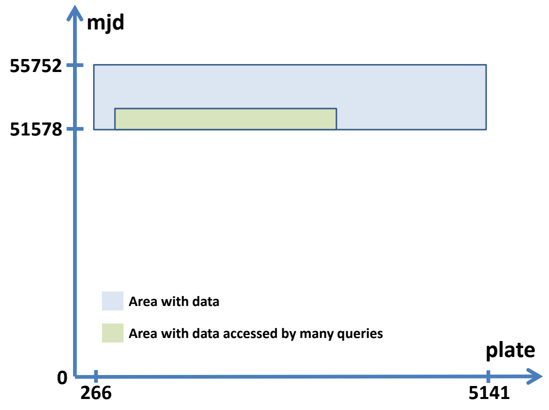
Data spaces.

The data space of a relation $R \in \mathbf{DB}$ which consists of columns $a_1, \ldots, a_t$ is defined as

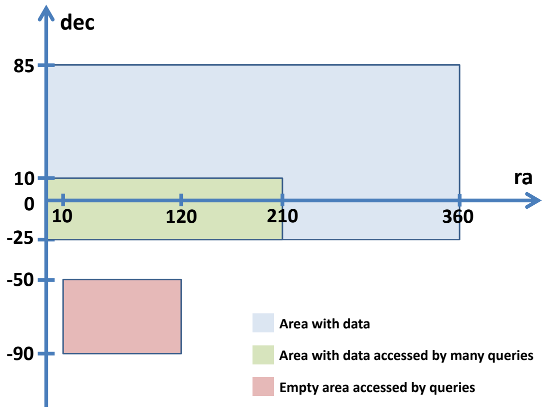
$$space(R) = dom(a_1) \times \ldots \times dom(a_t)$$

where $dom(a_i)$ is the domain of column a_i. In other words, the data space of R is the Cartesian product of the domains of its columns. Note that the data space of R is not confined to the database content. Further, when $t = 1$, i.e., R contains one column only, $space(R) = space(a_1) = dom(a_1)$. In the rest of this paper, unless specified otherwise, we use R to denote both the relation R and $space(R)$.

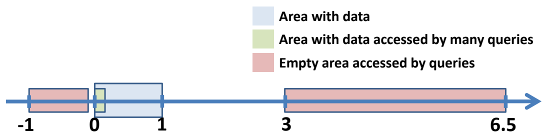
The data space of $\mathbf{DB}$ is defined to be the Cartesian product of all of its relations. The data space of each relation is a *subspace* of the data space of $\mathbf{DB}$.



(a) SpecObjAll.plate (smallint) vs. SpecObjAll.mjd (int)



(b) PhotoObjAll.ra (float) vs. PhotoObjAll.dec (float)



(c) Photoz.z (real)

Figure 1: Some access areas extracted from SkyServer query log.

Area of data content and empty area of a data space. Consider again relation R with columns $a_1, \ldots, a_t$. For each column a_i, if a_i is numerical, we let $content(a_i)$ be the minimum bounding box of its data values. If a_i is instead categorical, we let $content(a_i)$ be the set of values of a_i.

The area of data content of R is defined as

$$content(R) = content(a_1) \times \ldots \times content(a_t) \quad ,$$

i.e., the minimum bounding hyper-rectangle of the data content.

The empty area of $space(R)$ is $empty(R) = space(R) \setminus content(R)$. Roughly speaking, $empty(R)$ is the part of $space(R)$ which is not occupied by any database object. Note that this is a conservative way of estimating $empty(R)$. A more stringent notion could help to identify larger empty areas accessed by users.

Atomic predicates. Atomic predicates are those without deeper propositional structure. Though having many forms, we focus on atomic predicates having the form of "$a\ \theta\ c$" where a is a database column, c is a constant, and θ is either $<, \leq, =, >, \geq$, or $<>$. We name this type of predicate as *column-constant* atomic predicate.

Queries. Consider a query $\mathbf{q}$ in the log, which is issued against a state $\mathcal{T}$ of database $\mathbf{DB}$. Typically, $\mathbf{q}$ consists of SELECT, FROM, and possibly WHERE, GROUP BY, HAVING and ORDER BY clauses. The ORDER BY clause is not relevant for our purpose since it does not influence which parts of the data space actually are accessed. Thus, from now on, we exclude the ORDER BY clause.

Assume that $\mathbf{q}$ accesses relations $\mathcal{R} = \{R_1, \ldots, R_N\}$ where $R_i \neq R_j$ if $i \neq j$. Note that this excludes self-joins, which do not occur in the SkyServer query log that we considered. This type of join typically results in different aliases of the same relation and causes unreliable comparison of similarity between queries [4]. Each relation $R_i \in \mathcal{R}$ can appear either in the FROM clause or in any other clause, possibly in embedded subqueries. The area of $\mathbf{DB}$ accessed by $\mathbf{q}$ is a subset of $R_1 \times \cdots \times R_N$, which is called the universal relation of $\mathbf{q}$:

DEFINITION 1. *The universal relation of* $\mathbf{q}$ *is defined as:*

$$\mathcal{U} = R_1 \times \cdots \times R_N \quad .$$

The set of tuples of $\mathcal{U}$ *at state* $\mathcal{T}$ *of* $\mathbf{DB}$ *is denoted as* $(\mathcal{U}, \mathcal{T})$.

Each output column of $\mathbf{q}$ may or may not belong to the relations accessed, since new output columns may be created, e.g., columns holding constant values. We denote the set of columns appearing in the WHERE clause of $\mathbf{q}$ as $\mathcal{A}_W$, the ones in the GROUP BY clause as $\mathcal{A}_G$, the ones in the HAVING clause as $\mathcal{A}_H$, and the ones in any (nested query of) other clauses as $\mathcal{A}_S$. We note that $\mathcal{A}_W$, $\mathcal{A}_G$, and $\mathcal{A}_H$ may be empty. We define $\mathcal{A} = \mathcal{A}_W \cup \mathcal{A}_G \cup \mathcal{A}_H \cup \mathcal{A}_S$.

The WHERE clause can contain one or more atomic predicates on the columns of some relation(s) in $\mathcal{R}$. Likewise, the HAVING clause can contain one or more atomic predicates on the columns of some relation(s) in $\mathcal{R}$, typically in combination with aggregate functions like SUM, AVG, etc. Additional predicates can appear in (nested queries of) any other clause. Together, all these predicates and their connections form a constraint on $\mathcal{U}$, in the form of a Boolean expression. We refer to this Boolean expression as $\mathcal{P}$. Note that $\mathcal{P}$ is independent of any database state. As shown in Section 4, it is not always easy to extract $\mathcal{P}$ from $\mathbf{q}$. Instead, this depends on the type of $\mathbf{q}$ as well as on the definition of access area. For now, we simply use $\mathcal{P}$ for the sake of exposition. We have:

DEFINITION 2. *The result set of* $\mathbf{q}$ *at state* $\mathcal{T}$ *is the tuples of* $(\mathcal{U}, \mathcal{T})$ *which satisfy* $\mathcal{P}$. *We denote it as* $(\mathcal{U}, \mathcal{T})_{\mathcal{P}}$.

Following Definition 2, we have $(\mathcal{U}, \mathcal{T})_{\text{TRUE}} = (\mathcal{U}, \mathcal{T})$.

2.2 Naïve Definitions of Access Area

At first sight, one can define the access area of a query **q** to be either (a) the area of the data space covered by its result set, or (b) the part of $\mathbf{DB}$ accessed during the execution of **q**, as follows.

Option (a).

The access area of **q** is its result set at state $\mathcal{T}$, i.e., $(\mathcal{U}, \mathcal{T})_\mathcal{P}$. Under this scheme, an access area can be empty when the user is interested in an area where no object exists. Using this definition, we could define the access area of **q** as the minimum bounding box of $(\mathcal{U}, \mathcal{T})_\mathcal{P}$. Typically, $(\mathcal{U}, \mathcal{T})_\mathcal{P}$ is not available in the query log. This means that we have to re-issue **q** against the database, most likely at a different state $\mathcal{T}'$. So this definition suffers from two drawbacks. First, we may lose important information on the constraint $\mathcal{P}$ that the user had defined, as two very different queries can share the same result set (or minimum bounding box) independently of their constraints, i.e., their original intents. Second, if $\mathcal{T}' \neq \mathcal{T}$, it may be the case that $(\mathcal{U}, \mathcal{T}')_\mathcal{P} \neq (\mathcal{U}, \mathcal{T})_\mathcal{P}$. Thus, this definition is not meaningful for our purpose.

Option (b).

The access area of **q** is the part of $\mathbf{DB}$ at state $\mathcal{T}$ which has been accessed during the execution of **q**. In general, the query engine determines the part of the database to be accessed. To accomplish this, it relies on different statistics, e.g., query workload, network statistics, to decide on an execution plan for the query. In consequence, this definition of access areas leaves them dependent on factors that do not pertain to queries themselves. This is undesirable for capturing intents of users. Further, it is also difficult to impossible to compute access areas with an off-the-shelf commercial DBMS.

2.3 Our Definition of Access Area

DEFINITION 3. *Let a database state $\mathcal{T}$ of $\mathbf{DB}$ allowed by the database schema be given. A tuple $t \in \mathcal{U}$ is said to **influence** the result set $(\mathcal{U}, \mathcal{T})_\mathcal{P}$ of **q** iff:*

$$(\mathcal{U} \setminus \{t\}, \mathcal{T})_\mathcal{P} \neq (\mathcal{U}, \mathcal{T})_\mathcal{P} \quad .$$

That is, if t is removed from $\mathcal{U}$, the result set of **q** at state $\mathcal{T}$ will change. Our definition of access area now is as follows:

DEFINITION 4. *The access area of **q** is:*

$$\{t \in \mathcal{U} : \exists \mathcal{T} \text{ allowed by } \mathbf{DB} \text{ s.t. } t \text{ influences } (\mathcal{U}, \mathcal{T})_\mathcal{P}\} \quad .$$

That is, the access area of **q** is given by the set of all tuples contained in the universal relation $\mathcal{U}$ that influence the result set of **q** in **some** state allowed by the database schema. Following Definition 4, the access area of a query **q** represents the part of the data space the user is interested in, without limiting it to a certain state. For instance, let us consider the following example query:

SELECT $*$
FROM T
WHERE u **BETWEEN** 1 **AND** 8

According to Definition 4, the access area of this query is $\sigma_{u \geq 1 \wedge u \leq 8}(T)$, even if $u \notin [1, 8]$ for all tuples of relation T in its current state. This is because, for any tuple that satisfies $1 \leq u \leq 8$, a database state can exist, allowed by the schema, such that the tuple influences the result.

Therefore, our definition of access area copes with queries that do not return any row as well as with those that resulted in execution errors, e.g., *"Maximum 60 queries allowed per minute"* or *"limit is top 500000"* in the SkyServer scenario. This is crucial as it makes sense to capture what the users intended to access, independent of the actual database content and load constraints.

2.4 Extraction of Access Areas

To extract the access area of **q**, we have to obtain $\mathcal{U}$ and $\mathcal{P}$. While extracting $\mathcal{U}$ is rather simple, extracting $\mathcal{P}$ is more intricate. In fact, it is not always possible to extract $\mathcal{P}$ exactly. When this is the case, we derive a Boolean expression in conjunctive normal form which approximates $\mathcal{P}$. With the derivation of $\mathcal{P}$ or its approximation, we typically transform **q** to an **intermediate** format. In particular, concurrently with identifying the access area of **q**, we transform **q** to a query **q'** as follows:

SELECT $*$
FROM $R_1, R_2, \ldots, R_N$
WHERE
$F(p_1(a_{1,1}, a_{2,1}, \ldots, a_{T_1,1}), \ldots, p_K(a_{1,K}, a_{2,K}, \ldots, a_{T_K,K}))$

where

- $\{a_{1,1}, \ldots, a_{T_K,K}\} \subset \mathcal{A}$,
- p_i for every $i \in [1, K]$ being an atomic predicate defined on columns $\{a_{1,i}, a_{2,i}, \ldots, a_{T_i,i}\}$, and *derived* from $\mathcal{P}$,
- F being a *conjunctive normal form* of the atomic predicates $\{p_1, \ldots, p_K\}$, i.e., a conjunction of disjunctions. For instance, assume that $K = 4$, $F(p_1, p_2, p_3, p_4)$ could be $(p_1 \vee p_2) \wedge (p_3 \vee p_4)$.

We note that the WHERE clause is optional and can be empty.

With the above transformation, the access area of query **q'** is the one of **q**, which is $\sigma_{F(p_1,\ldots,p_K)}(R_1 \times R_2 \times \cdots \times R_N)$, or $\sigma_{F(p_1,\ldots,p_K)}(\mathcal{U})$.

For illustration, the following query is already in intermediate format:

SELECT $*$
FROM T
WHERE (T.u <= 5 **OR** T.u >= 10) **AND** T.v <= 5

So no transformation is required and we can obtain its exact access area easily.

However, in practice, an exact transformation is not always straightforward for most query types, especially nested queries in combination with (NOT) IN, (NOT) EXISTS, (NOT) ALL and (NOT) ANY. Instead, sophisticated analysis is required. More details are in Section 4. Having introduced our notion of access area, we discuss related work in the next section and point out why it is not suited to address the problem.

3. RELATED WORK

Processing and extracting useful information from SQL queries has been studied for some time. In this paper, we divide related work into the following categories: extracting information from queries, processing query logs, and distance measures for queries.

3.1 Extracting Information from Queries

Ceri and Gottlob [6] studied transforming SQL queries into relational algebra. Their goal is to preserve the constraints defined by typical SQL structures such as EXISTS, IN, and aggregate functions. However, since they did not focus on extracting access areas, they have not introduced methods to convert complex Boolean expressions, such as those involved in set operations (with IN, ANY, etc.), to simple ones with atomic predicates. Such simple Boolean expressions in turn are required to constrain the data space, and hence, for the extraction of access areas.

Parsing, relaxing and rewriting of queries are investigated in [9, 14, 18, 21]. Their primary goal is to either point out logical flaws in query sub-expressions, rewrite queries into an optimized, more declarative form that avoids empty results and improves query performance. Thus, they do not address the problem studied here.

A more intuitive way of representing queries is reported in [11]. The meaning (or intent) of queries is captured by an UML-like notation. This notation breaks down the query structure to show relationships between different fragments. Yet, the proposed method does not have a mechanism to further reduce complex fragments, such as EXISTS, to simple ones.

There also is related work to information extraction of queries from fields such as Natural Language Processing (NLP), Information Retrieval (IR), and Machine Learning (ML). For example, [15] and [19] propose to transform SQL queries to a different format—the one of natural language. In particular, queries are represented by graphs using a query template mechanism. The critical elements of SQL queries are extracted, and different strategies are presented to construct textual query descriptions from these elements. However, these studies also do not target at simplifying Boolean expressions to facilitate the extraction of access areas.

3.2 Processing Query Logs

Due to the vast amount of data produced by search engines and the pervasive tracking of user click-streams, researchers have studied query log processing to some extent. For instance, [17] and [22] aim at furthering our understanding of the behavior of users through their information-seeking activities. These articles demonstrate why query log processing is useful for data analytics. In more details, [17] transforms web logs into a format suitable for importing into databases. In addition, it builds a data warehouse from these cleaned and structured log files and provides an ad-hoc tool for analytic queries on this warehouse. [22] explores different statistics, which can be drawn from query logs such as query popularity, term popularity, average query length, and distance between repetitions of queries. [3, 4] on the other hand target OLAP logs with the objective to compare different users sessions.

Singh et al. [23] give a detailed analysis of the first five years since SDSS SkyServer went on-line. They clean and normalize web and SQL log files over several months, resulting in data structures including IP Name, Sessions, and Templates (skeleton SQL templates). They further compare SQL queries using fragments, N-grams, and the Jaccard Coefficient, and categorized the queries into those issued by bots or by mortals.

QueRIE, a query recommendation system [2, 8, 7, 20], is designed to work directly with SkyServer query logs. It uses two different approaches to achieve high accuracy in recommending new and interesting queries to users. The first approach extracts the most important SQL query fragments, while the second one uses the tuples a query retrieves.

SDSS Log Viewer [26] visualizes the SQL queries from the log files. To achieve the visualization, the author develops a process to transform the SkyServer SQL log files into a tabular format that can be stored in a database. Each query is tokenized in order to apply a visual encoding scheme and to extract the critical fragments. Depending on these fragments, queries are classified into four categories representing the type of "sky area" a query accesses: Rectangular Sky Area, Circular Sky Area, Single Point/Object, and others. Besides this, depending on the intention of the user, the author creates three categories for the major sets of queries: Scan Queries, Search Queries, and Retrieve Queries.

All of the above approaches do not solve the problem we study as they (a) neglect the notion of access area in each query, and (b) lack mechanisms to map queries to access areas.

3.3 Distance Measures for Queries

Existing distance measures for queries model them as either strings [25], feature vectors [1, 2, 12], sets of fragments [13], or graphs [24]. With all these representations, clustering is expected to be feasible. But these clusters either do not represent specific parts of the data space, or the mapping would need to be specified in the first place. In contrast to these, we focus on a similarity notion that allows us to aggregate a set of overlapping access areas.

4. REALIZATION

Different types of queries need different types of predicate extraction, i.e., different mappings to their access areas. Further, as we will explain in this section, for some types, non-trivial analysis is required to extract their access areas. As the SQL grammar has a high complexity, and as many variations exist between different database management systems, it lies beyond the scope of this paper to cover every possible valid SQL statement. Instead, we focus on the log file of one large database and extract the access areas from the statements in this log file. We consider $12,375,426$ SkyServer queries issued by users from 127 countries, starting from April 2012. We note that since the number of queries considered is large and their origins are diverse, we do not expect any major gaps when working with another public database or returning to SkyServer in the future. Further, our model can always be extended later to include yet unhandled types of queries or types of predicates, schema specific functions, as well as different SQL dialects. It is also possible to extract the information from an incoming stream of logged queries, to detect changes in this data stream and to notify the system operator about the occurrence of new predicates and query types.

For our system implementation, we use the log file from the 9^{th} data release of SDSS, which has been the latest release when we started this project. Currently, we classify the queries logged into several categories to facilitate processing. Nevertheless, we apply the same procedure for any query type. That is, following Section 2.4, before extracting the access area of a query, we transform it into a query of the intermediate format, if necessary. Then we process the transformed query to obtain the access area. The details are as follows.

4.1 Simple Queries

Queries belonging to this category are those without (a) joins, (b) GROUP BY and HAVING clauses, and (c) nested subqueries. In other words, each query $\mathbf{q}$ of this type either already is of the intermediate format or can be straightforwardly converted to the intermediate format (which involves in converting its constraint $\mathcal{P}$ to a conjunctive normal form). For each simple query, since its predicates can be extracted exactly, we can easily obtain its exact access area. For example, the following query

```
SELECT u
FROM T
WHERE u >= 1 AND u <= 8 AND s > 5
```

is a simple query with access area $\sigma_{u\geq 1\wedge u\leq 8\wedge s>5}(T)$.

From an implementation point of view, we need special handling of queries containing specific operators, namely BETWEEN and NOT. In particular, for each predicate with a BETWEEN operator, we need to derive two new predicates to replace the original one. For example, $T.u$ *BETWEEN* 5 *AND* 10 is converted to

$T.u \geq 5 \wedge T.u \leq 10$. For predicates containing the NOT operator, we transform them by inverting the respective predicate. For example, *NOT* $(T.u > 5 \wedge T.v \leq 10)$ becomes $T.u \leq 5 \vee T.v > 10$.

4.2 Join Queries

Most types of JOINs can be converted simply by keeping the relation and pushing any join condition to the WHERE clause (CROSS JOIN, INNER JOIN, EQUI JOIN, NATURAL JOIN). Other types need further consideration, as follows.

EXAMPLE 2. *Consider a query **q** with FULL OUTER JOIN:*

```
SELECT *
FROM T FULL OUTER JOIN S ON (T.u = S.u)
```

Here, the relations involved are T and S, i.e., $\mathcal{U} = T \times S$. *Regarding the constraint on* $\mathcal{U}$, *we observe that FULL OUTER JOIN keeps all tuples of both relations, even if there is no match for* $T.u = S.u$. *Thus, any tuple in* $T \times S$ *can influence the result set of the original query. That is, there is no constraint on* $\mathcal{U}$. *Hence, we convert the query to:*

```
SELECT *
FROM T, S
```

The access area then is $\sigma(T \times S)$.

From Example 2, we see that identifying $\mathcal{P}$ goes beyond simply extracting the predicates as-is.

EXAMPLE 3. *Consider a query **q** with RIGHT OUTER JOIN:*

```
SELECT *
FROM T RIGHT OUTER JOIN S ON (T.u = S.u)
```

Again, we have $\mathcal{U} = T \times S$. *However, identifying* $\mathcal{P}$ *is more complex. We observe that this query returns all tuples in S, together with those tuples from T which match at least one tuple in S. That is, it is equivalent to:*

```
SELECT *
FROM T, S
WHERE T.u IN (SELECT S.u FROM S)
```

The above query is nested, and we have a special procedure to handle it, which comes in Section 4.4. Queries with LEFT OUTER JOIN are handled analogously.

4.3 Aggregate Queries

There is a variety of aggregate queries in practice, too many to be covered fully. Thus, we confine our study to aggregate queries that can be found in the SkyServer query log. Such queries have the following form:

```
SELECT *
FROM [...]
WHERE [...]
GROUP BY [...]
HAVING AGG(a) [< | ≤ | = | > | ≥ | <>] c
```

That is, the HAVING clause is of the form $AGG(a)\ \theta\ c$ where a is a column and θ is either $<, \leq, =, >, \geq$, or $<>$. Further, c is a constant. The FROM clause can consist of any number of relations. In addition, the WHERE clause is a conjunctive normal form of atomic predicates. The GROUP BY clause in turn can consist of any combination of columns. Here, GROUP BY and WHERE clauses are optional. We consider the aggregate functions SUM(), COUNT(), MIN(), MAX(), and AVG() in our implementation, though we note that MAX() does not appear in the log.

Overall, we find an exact transformation that preserves access areas for queries of the above format. As a representative, we illustrate our handling of such queries for the SUM function. To this end, we observe that there are several scenarios; some of which are shown below. The others are in [5].

LEMMA 1. *Consider the following query:*

```
SELECT T.u, SUM(T.v)
FROM T
GROUP BY T.u
HAVING SUM(T.v) > c
```

where c is a constant. Assume that $dom(T.v) = [inf, supp]$ *where supp could be* $+\infty$, *and inf could be* $-\infty$. *We have:*

- *If* $supp > 0$, *the access area is* T.
- *If* $supp \leq 0 \wedge c > supp$, *the access area is* $\emptyset$.
- *If* $supp \leq 0 \wedge c \in dom(T.v)$, *the access area is* $\sigma_{T.v>c}(T)$.
- *If* $supp \leq 0 \wedge c < inf$, *the access area is* T.

PROOF. Consider an arbitrary tuple $t \in T$ (i.e., t is a tuple in the data space of T).
Case 1: $supp > 0$. Let k be an integer such that

$$k > \left\lceil \frac{2(c - t.v)}{supp + \max\{inf, 0\}} \right\rceil \quad .$$

Consider a database state in which T contains t and k other tuples $\{t'_1, \ldots, t'_k\}$; each tuple t'_i satisfies that $t'_i.u = t.u$ and $t'_i.v = \frac{supp+\max\{inf,0\}}{2}$. Since $t'_i.v \in dom(T.v)$, this state is allowed by the database schema. We have: $t.v + \sum_{i=1}^{k} t'_i.v > c$. Thus, t influences the result set, i.e., the access area is T.
Case 2: $supp \leq 0$. We have the following cases:

- $c > supp$: For every $x' \in dom(T.v)$, it holds that: $t.v + x' \leq t.v < c$. This implies that t can never be part of the access area. Thus, the access area is $\emptyset$.
- $c \in dom(T.v)$: If $t.v > c$, we construct a database state in which T contains t only, which conforms to the database schema. Further, t influences the result set. In contrast, if $t.v \leq c$, we can deduce that t cannot influence the result set. Therefore, the access area is $\sigma_{T.v>c}(T)$.
- $c < inf$: The access area is T.

This concludes our proof. □

In the following lemmas, for simplicity, we assume that the domain of each column involved is large enough such that with respect to its data type, it can be considered as $(-\infty, +\infty)$. This holds in general since queries tend to not have predicates containing values near the bounds of domains.

LEMMA 2. *Consider the following query:*

```
SELECT T.u, SUM(T.v)
FROM T
WHERE T.v < c1
GROUP BY T.u
HAVING SUM(T.v) > c2
```

where c_1 *and* c_2 *are constants. We have:*

- *If* $c_1 > 0$*, the access area is* $\sigma_{T.u<c_1}(T)$*.*
- *If* $c_1 \leq 0$ *and* $c_2 \geq 0$*, the access area is* $\emptyset$*.*
- *If* $c_1 \leq 0$ *and* $c_2 < 0$*: If* $c_2 < c_1$*, the access area is* $\sigma_{T.u<c_1 \wedge T.u>c_2}(T)$*. Otherwise, it is* $\emptyset$*.*

PROOF. Consider an arbitrary tuple $t \in T$ (i.e., t is a tuple in the data space of T). If $t.v \geq c_1$, $t.v$ is not part of the access area. Hence, we will only consider the case where $t.v < c_1$.

$c_1 > 0$*:* Let k be an integer such that $k > \left\lceil \frac{2(c_2 - t.v)}{c_1} \right\rceil$. Consider a database state in which T contains t and k other tuples $\{t'_1, \ldots, t'_k\}$ where $t'_i.u = t.u$ and $t'_i.v = \frac{c_1}{2}$. Since $\frac{c_1}{2}$ is a valid value of $T.v$, this state is allowed by the database schema. We have: $t.v + \sum_{i=1}^{k} t'_i.v > c_2$. Thus, the access area is: $\sigma_{T.v<c_1}(T)$.

$c_1 \leq 0$ *and* $c_2 \geq 0$*:* For every $x' < c_1$, it holds that: $t.v + x' \leq t.v < c_1 \leq c_2$. This implies that t can never be part of the access area. Thus, the access area is $\emptyset$.

$c_1 \leq 0$ *and* $c_2 < 0$*:* Consider an arbitrary $x' < c_1$. If $t.v \leq c_2$, we have: $t.v + x' < t.v \leq c_2$, i.e., t is not part of the access area. As a result, if $c_2 < c_1$, the access area is: $\sigma_{T.u<c_1 \wedge T.u>c_2}(T)$. Otherwise, it is $\emptyset$. □

LEMMA 3. *Consider the following query:*

```
SELECT T.u, SUM(T.v)
FROM T
WHERE T.v > c1
GROUP BY T.u
HAVING SUM(T.v) > c2
```

where c_1 *and* c_2 *are constants. The access area of this query is* $\sigma_{T.u>c_1}(T)$*.*

PROOF. Consider an arbitrary tuple $t \in T$ (i.e., t is a tuple in the data space of T) where $t.v > c_1$.

$c_1 > 0$*:* Let $k > \left\lceil \frac{c_2 - t.v}{c_1} \right\rceil$. We have: $k \cdot c_1 > c_2$. Analogously to Lemma 2, the access area is: $\sigma_{T.u>c_1}(T)$.

$c_1 \leq 0$*:* Let $k > \lceil c_2 - t.v \rceil$. Consider a database state in which T contains t and k other tuples $\{t'_1, \ldots, t'_k\}$ where $t'.u = t.u$ and $t'.v = 1$. This state is allowed by a database schema. In addition, $t.v + \sum_{i=1}^{k} t'_i.v > c_2$. Thus, the access area still is: $\sigma_{T.u>c_1}(T)$. □

In our implementation, we have covered all cases for the SUM function. For each query with $\mathrm{SUM}(a)\ \theta\ c$ in the HAVING clause, we check if a belongs to some relation in the FROM clause. If it does not, we ignore it. Otherwise, we apply special mappings. The above cases are examples of such mappings. The detailed handling of other cases as well as other aggregate functions is in [5]. There we also confine things to one aggregate function per HAVING clause. This is not a problem in reality, as the more general case does not occur in the SkyServer query log at all.

4.4 Nested Queries

Queries of this type manifest themselves either explicitly with operators such as EXISTS, IN, ANY, ALL, or implicitly in some nested predicate, e.g., *T.u = (SELECT S.u FROM S WHERE S.v = 12)*. As with aggregate queries, we do not discuss every possible aspect of nested queries and refer to [5] for further information. Instead, we focus on several issues, and confine the presentation here to the EXISTS operator. To keep the exposition simple, we discuss nested queries of the following form, which covers all nested queries appearing in the log:

```
SELECT *
FROM [...]
WHERE [...]
OPT EXISTS(q_1)
...
OPT EXISTS(q_m)
```

where OPT is either AND or OR, and $\mathbf{q}_i$ is a query of the intermediate format. Further, each $\mathbf{q}_i$ refers to one single relation, and this relation does not appear in the FROM clause of the parent query. This also avoids any implicit self-join.

Given a nested query $\mathbf{q}$ of the above format, we transform it into the intermediate format as follows:

- Group m subqueries in the EXISTS clauses based on the relations they refer to. W.l.o.g., let the groups be $G_1 = \{\mathbf{q}^1_1, \ldots, \mathbf{q}^1_{m_1}\}, \ldots, G_l = \{\mathbf{q}^l_1, \ldots, \mathbf{q}^l_{m_l}\}$, where $l \leq m$, $\mathbf{q}^u_v \in \{\mathbf{q}_1, \ldots, \mathbf{q}_m\}$, and $\sum_{u=1}^{l} m_u = m$.
- Let $\mathbf{q}_i.FROM$ be the relation in the FROM clause of $\mathbf{q}_i$. Further, we write $\mathbf{q}_i.WHERE$ for all predicates together with their connections in the WHERE clause of $\mathbf{q}_i$. We transform $\mathbf{q}$ to the following query:

  ```
  SELECT *
  FROM [...], q_1.FROM, ..., q_m.FROM
  WHERE [...]
  OPT (q^1_1.WHERE OR ... OR q^1_{m_1}.WHERE)
  ...
  OPT (q^l_1.WHERE OR ... OR q^1_{m_l}.WHERE)
  ```

- Subsequent simple transformations may be required to convert the constraint in the WHERE clause of the above query to a conjunctive normal form.

We present three categories of nested queries having the above format to show why our transformation preserves the access area of $\mathbf{q}$ exactly. In fact, all nested queries with the EXISTS operator occurring in the SkyServer query log fall into these three categories.

LEMMA 4. *Consider the following query:*

```
SELECT *
FROM T
WHERE T.u > α
AND EXISTS
(SELECT * FROM S
WHERE S.u = T.u AND S.v < β)
```

where α *and* β *are constants. The access area of this query is:* $\sigma_{T.u>\alpha \wedge S.u=T.u \wedge S.v<\beta}(T \times S)$*.*

PROOF. The access area of this query is a subset of $T \times S$. We prove that an arbitrary element $(t, s) \in T \times S$ influences the result of the query if and only if $t.u > \alpha$, $s.u = t.u$, and $s.v < \beta$.

($\Rightarrow$): Consider $(t, s) \in T \times S$ that influences the result. Then we have that $t.u > \alpha$. Next, if $s.v \geq \beta$, we can always remove (t, s) without influencing the result. So it must hold that $s.v < \beta$. Further, if there does not exist any $t' \in T$ such that $t'.u > \alpha$ and $s.u = t'.u$, then again we can safely remove (t, s). Thus, such a t' must exist. If $s.u \neq t.u$, as long as we keep (t', s), we can safely remove (t, s) while the result is not impacted. Hence, it holds that $s.u = t.u$. Thus, we have $t.u > \alpha$, $s.u = t.u$, and $s.v < \beta$.

($\Leftarrow$): Let (t, s) be such that $t.u > \alpha$, $s.u = t.u$, and $s.v < \beta$. We construct a database state where T contains only t, and S

contains only s. Clearly, if we remove (t, s), the result of the query in this database state is changed.

Combining $(\Rightarrow)$ and $(\Leftarrow)$, we conclude our proof. Using our procedure, we have $m = 1$, $\mathbf{q}_1.FROM$ is S, and $\mathbf{q}_1.WHERE$ is *S.u = T.u AND S.v* $< \beta$. Thus, the transformed query is:

```
SELECT ∗
FROM T, S
WHERE T.u > α AND S.u = T.u AND S.v < β
```

which is as expected. □

LEMMA 5. *Consider the following query:*

```
SELECT ∗
FROM T
WHERE T.u > α
AND EXISTS
(SELECT ∗ FROM S
WHERE S.v < β AND S.u = T.u)
AND EXISTS
(SELECT ∗ FROM S
WHERE S.v >= γ AND S.u = T.u)
```

where α, β, and γ are constants, and $\gamma \geq \beta$. The access area of this query is: $\sigma_{T.u>\alpha \wedge S.u=T.u \wedge (S.v<\beta \vee S.v\geq\gamma)}(T \times S)$.

PROOF. We prove that $(t, s) \in T \times S$ influences the result of the query if and only if: $t.u > \alpha \wedge s.u = t.u \wedge (s.v < \beta \vee s.v \geq \gamma)$.

$(\Rightarrow)$: If (t, s) influences the result, we have $t.u > \alpha$. If $\beta \leq s.v < \gamma$, we can safely discard (t, s) without influencing the result. Hence, it must hold that $s.v < \beta \vee s.v \geq \gamma$. By reasoning similarly to the proof of Lemma 4, we have $s.u = t.u$. Thus, if (t, s) influences the result, then: $t.u > \alpha \wedge s.u = t.u \wedge (s.v < \beta \vee s.v \geq \gamma)$.

$(\Leftarrow)$: Let (t, s) be such that $t.u > \alpha$, $s.u = t.u$, and $s.v < \beta \vee s.v \geq \gamma$. W.l.o.g., we assume that $s.v < \beta$. We construct a database state where T contains t only, and S contains s, and another s', where $s'.u = t.u$ and $s'.u \geq \gamma$. Then, if we remove (t, s), the query result in this database state is changed.

Using our procedure, we transform the original query to:

```
SELECT ∗
FROM T, S
WHERE T.u > α AND S.u = T.u
AND (S.v < β OR S.v >= γ)
```

which preserves the access area exactly. □

LEMMA 6. *Consider the following query:*

```
SELECT ∗
FROM T
WHERE T.u > α
OR EXISTS
(SELECT ∗ FROM S
WHERE S.v < β AND S.u = T.u)
OR EXISTS
(SELECT ∗ FROM S
WHERE S.v >= γ AND S.u = T.u)
```

where α, β, and γ are constants, and $\gamma \geq \beta$. The access area of this query is: $\sigma_{(T.u>\alpha \vee S.u=T.u)\wedge(T.u>\alpha \vee S.v<\beta \vee S.v\geq\gamma)}(T \times S)$.

PROOF. Following a proof similar to the one of Lemma 6, we derive that $(t, s) \in T \times S$ influences the result of the query if and only if $t.u > \alpha$, or $s.u = t.u \wedge (s.v < \beta \vee s.v \geq \gamma)$. Converting this Boolean expression to a conjunctive normal form, we arrive at the result. □

We can also use the three categories of nested queries discussed so far to extract access areas of nested queries that have more than one nested level, i.e., we are able to generalize beyond the query log of SkyServer. The following example illustrates our point.

EXAMPLE 4. *Consider the following query:*

```
SELECT ∗
FROM T
WHERE T.u > α
AND EXISTS
(SELECT ∗ FROM S
WHERE S.u = T.u AND S.v < β
AND EXISTS
(SELECT ∗ FROM R
WHERE R.v = S.v AND R.x < γ))
```

where α, β, and γ are constants. From Lemma 4, we know that in term of access area, the subquery of the outer EXISTS *operator is equivalent to:*

```
SELECT ∗
FROM S, R
WHERE S.u = T.u AND S.v < β
AND R.v = S.v AND R.x < γ
```

Here, we temporarily consider $T.u$ as a constant. With this transformation, the original query now has only one nested level. Continuing to apply Lemma 4, we transform the original query to:

```
SELECT ∗
FROM T, S, R
WHERE T.u > α AND S.u = T.u AND S.v < β
AND R.v = S.v AND R.x < γ
```

In addition, we can also process nested queries with aggregate subqueries by combining the theories of this section and of Section 4.3. Furthermore, we have an approximation scheme to process complex nested queries in general, i.e., the ones that do not conform to any format already discussed. The details of this approximation scheme and our handling of nested queries with other operators are in [5].

4.5 System Implementation

We now briefly describe our actual implementation. For a given query, we first parse it to identify its single fragments. We use JSqlParser[1], as it is an open source project under the LGPL license and has a powerful, extensible grammar that supports most of the SQL structure occurring in the SkyServer logs. Second, we transform these fragments into a form that conforms to our intermediate format (see Section 2.4). In particular, we extract the relations the query addresses, including any relation in any nested query, and the constraints on these relations and related columns. Third, we convert the constraints derived into conjunctive normal form. Finally, as a cleanup step, we replace any remaining alias with the real name of the relation and order the list of relations alphabetically. Besides this, we perform some consolidation on the remaining predicates: We remove redundant constraints, merge overlapping constraints, and check the set of constraints for contradictions.

5. OUR DISTANCE FUNCTION

Our end goal is to discover interesting access areas in the data space that may represent the user interests. To accomplish this, we

[1] `http://jsqlparser.sourceforge.net/home.php`

need to extract a bigger picture out of the access areas of similar queries. In other words, we need a procedure to aggregate the access areas of a large set of queries. We aim at achieving this by clustering queries based on overlap as our main objective of similarity. The distance measure that we use for clustering simply quantifies the overlap (i.e., the similarity of access areas). Please note that other distance measures could be used for this purpose. The distance does not even have to be a metric [16]. However, it should have its main focus on the content of queries and not their structure like in other cases [4]. As part of our aggregation, we define such a distance function as follows.

Consider two queries $\mathbf{q}_1$ and $\mathbf{q}_2$ of intermediate form (see Section 2.4). We define their distance as follows:

$$\begin{aligned} d(\mathbf{q}_1, \mathbf{q}_2) = d_{tables}(\mathbf{q}_1.FROM, \mathbf{q}_2.FROM) \\ + d_{conj}(\mathbf{q}_1.WHERE, \mathbf{q}_2.WHERE) \end{aligned} \quad (1)$$

where $\mathbf{q}.FROM$ denotes the tables in the access area of query $\mathbf{q}$, and $\mathbf{q}.WHERE$ denotes its WHERE part, which is its access area (in a conjunctive normal form). Note that with proper instantiation of d_{tables} and d_{conj}, one could actually compute this distance function on the raw queries as in [4]. However, we have shown that properly extracting access areas of queries is far from simply using as-is all predicates appearing in the queries. Instead, one needs to resort to our transformation of queries to intermediate format. In the followings, we provide our instantiation of d_{tables} and d_{conj}.

5.1 Distance of Access Tables d_{tables}

We use the Jaccard coefficient to measure the distance between the two sets of table names $\mathbf{q}_1.FROM$ and $\mathbf{q}_2.FROM$:

$$\begin{aligned} & d_{tables}(\mathbf{q}_1.FROM, \mathbf{q}_2.FROM) \\ & = 1 - \frac{|\mathbf{q}_1.FROM \cap \mathbf{q}_2.FROM|}{|\mathbf{q}_1.FROM \cup \mathbf{q}_2.FROM|} \end{aligned} .$$

The Jaccard coefficient has the disadvantage that corner cases have to be defined if both queries do not access any table. This may occur if a query only queries database constants. In this case, we set d_{tables} to 0.

5.2 Distance of Conjunctions d_{conj}

Consider two Boolean expressions b_1 and b_2 which are both in conjunctive normal form. We define their distance to be:

$$\begin{aligned} & d_{conj}(b_1, b_2) \\ & = \frac{\sum\limits_{o_1 \in b_1} \min\limits_{o_2 \in b_2} d_{disj}(o_1, o_2) + \sum\limits_{o_2 \in b_2} \min\limits_{o_1 \in b_1} d_{disj}(o_1, o_2)}{|b_1| + |b_2|} \end{aligned}$$

where each $o_1 \in b_1$ is a disjunction of Boolean expression(s), and $|b_1|$ is the number of disjunctions of b_1. We define each $o_2 \in b_2$ and $|b_2|$ similarly. In addition, $d_{disj}(o_1, o_2)$ is the distance between o_1 and o_2, which is given by:

$$\begin{aligned} & d_{disj}(o_1, o_2) \\ & = \frac{\sum\limits_{p_1 \in o_1} \min\limits_{p_2 \in o_2} d_{pred}(p_1, p_2) + \sum\limits_{p_2 \in o_2} \min\limits_{p_1 \in o_1} d_{pred}(p_1, p_2)}{|o_1| + |o_2|} \end{aligned}$$

where $p_1 \in o_1$ is an atomic predicate, and $|o_1|$ is the number of atomic predicates of o_1. We define each $p_2 \in o_2$ and $|o_2|$ similarly. The distance between two atomic predicates p_1 and p_2, given by $d_{pred}(p_1, p_2)$, is as follows:

p_1 and p_2 refer to the same single column. This means that they are column-constant atomic predicates (see Section 2.1). We refer to the column as a.

First, we assume that a is numerical. Let $MBR(a)$ be the minimum bounding box of the area of $dom(a)$ that are accessed by all queries in the log, including those having accessed the empty area of $dom(a)$. Further, let $access(a) = content(a) \cup MBR(a)$. Since a typically has a data type, $dom(a)$ and hence $access(a)$ are intervals with finite bounds. We set $d_{pred}(p_1, p_2) = \frac{\text{overlap of intervals}}{\text{width of } access(a)}$, i.e., the normalized overlap of two respective intervals. For instance, assume that p_1 is $a < 3$, p_2 is $a > 2$, and $access(a_1) = [0, 5]$. We have $d_{pred}(p_1, p_2) = 1/5 = 0.2$. Here we use $access(a)$ instead of $content(a)$ for normalization to cope with queries accessing the empty area.

On the other hand, if a is categorical, we denote $access(a)$ as the union between $content(a)$ and the set of values of $dom(a)$ accessed by all queries in the log. Further, we replace the overlap of intervals by the number of items p_1 and p_2 have in common, and the width of $access(a)$ by its cardinality. Note again that since a typically has a data type, $dom(a)$ and hence $access(a)$ have finite numbers of values.

p_1 and p_2 refer to different columns. We set $d_{pred}(p_1, p_2)$ to the proportion of the joint space of the involved columns occupied by p_1 and p_2. For instance, assume that p_1 is $a_1 < 3$, p_2 is $a_2 > 2$, and $access(a_1) = access(a_2) = [0, 5]$. We have $d_{pred}(p_1, p_2) = (3 \times 3)/(5 \times 5) = 0.36$.

5.3 Implementation Issues

To use our distance function, for each column a, we need to know $access(a)$. Since $content(a) \subset access(a)$, we need to first identify $content(a)$. Normally, this can be done by simply querying the database. However, when doing this, we got the timeout error for many columns, especially those belonging to large relations. The issue can be resolved in two ways: (1) interact with the domain experts, or (2) estimate $content(a)$ and hence $access(a)$ from the database content. The advantage of (2) is that the system implementation does not need to be configured by hand/modified when turning to another database. On the other hand, the results with (1) might be better. However, our concern with this study is to find out whether our approach as a whole is practical, and whether results already are useful with relatively simple technical means. Optimizing output quality further is future work. This is why we have resorted to (2) in this current evaluation, as follows.

For each numerical column, we derive its statistics by querying a sample of its data, e.g., 100 rows, from SkyServer. Assume that $[m, M]$ is its value range obtained from the sample. Then, we set $access(a) = content(a) = \left[m - \frac{M-m}{2}, M + \frac{M-m}{2}\right]$, i.e., we double the size of the sampled range. When processing each query in the log containing a column-constant predicate of the form "a θ c", if it accesses data not falling into $access(a)$, we update this range accordingly.

For each categorical column, we do similarly. However, instead of the range, we maintain its set of values. If a query accesses a value that does not appear in this set, we update the set accordingly.

We stress again that our access to the SkyServer database at this point is only for this specific clustering purpose, and our extraction of access areas is not involved in any physical database access.

6. CASE STUDY: CLUSTERING TRANSFORMED DATA WITH DBSCAN

In this section, we present a case study where we cluster the transformed data using our distance function. Our objective is to find out if we can discover interesting aggregate access areas. Regarding clustering, there is a variety of existing algorithms in the literature. On the other hand, we want to find out whether results generated with relatively simple means are helpful from the perspective of a domain expert. Hence, we use a well-known, relatively simple, noise-aware algorithm, that does not need us to specify the number of clusters, namely DBSCAN [10].

6.1 The Data

The log originally contains $12,442,989$ queries. We were able to extract the access area of $12,375,426$ queries, which is more than 99.4%, leaving $67,563$ queries without extraction. The leftover queries are in fact not accepted by the grammar of the JSqlParser. This is because they (a) contain errors, (b) use user-defined SkyServer-specific functions, or (c) are not SELECT queries but statements with CREATE TABLE or DECLARE (issued by SkyServer administrators). So except for the pathological queries, our method performs well in extracting access areas.

6.2 Access Areas

In preliminary experiments, we have observed that DBSCAN (or, at least, its implementation used here) has severe performance problems when applied to the entire set of transformed queries. Thus, the following results are obtained on a (not necessarily representative) sample consisting of $5,611,087$ access areas of queries. Each access area in this sample is constrained to contain only predicates of the form either "$a\ \theta\ c$" (column-constant) or "$a_1\ \theta\ a_2$" (column-column). This is to increase interpretability of the results. Of course, while taking all queries into account might be more informative, using that subset does not contradict our objectives: We want to find hotspots of user interest, and we want to see how much overlap there is with the actual database content. We also want to learn how effective standard tools (e.g., an off-the-shelf clustering algorithm) in this context actually are.

In general, access areas of individual queries do not convey much information to the database owner, but a summary of this data for all queries is definitely interesting. To this end, using DBSCAN, we cluster the access areas in the sample described above. For each output cluster, we derive its minimum bounding hyper-rectangle, which we interpret as the aggregated access area of the queries involved. During this process, we leave out extreme range bounds by applying the 3-standard deviation rule. This is to ensure the robustness of the results. Overall, we obtain 403 clusters. We list 24 representative clusters in Table 1. We choose these clusters since we find them to contain few columns and hence, easy to interpret. For each cluster, we present the following information:

- Cluster ID.
- Cardinality: The number of access areas (queries) falling into the cluster.
- Area coverage: $\frac{v_{access}}{v_{content}}$ where v_{access} is the volume of the aggregated access area, and $v_{content}$ is the volume of the database content.
- Object coverage: $\frac{n_{access}}{n_{content}}$ where n_{access} is the number of objects falling into the aggregated access area, and $n_{content}$ is the number of objects of the database content.
- Access area: A Boolean expression describing the aggregated access area.

Going over the result, we find that most queries in each cluster are issued by different users, i.e., the cardinality of each cluster is approximately equal to the number of users. For each of the Clusters 1–17, its aggregated access area overlaps with a relatively small part of the database content. In particular, both of its area coverage and object coverage are fairly small (both less than 1% for Cluster 17). This shows that some users are interested in only a small part of the database content when issuing a query. We also see that while the area coverage is close to the object coverage in many clusters, this is not the case for Clusters 7, 8, 14, and 15. This is an indication that queries do not really follow the data distribution.

On the other hand, each of the Clusters 18–24 has its aggregated access area occupy empty area of the data space. Each such cluster contains from 17% (Cluster 20) up to 50% (Cluster 22) of the total number of queries accessing the same data space. This means that a significant number of queries refer to empty areas where no data objects are present.

6.3 Feedback from Domain Experts

This subsection describes qualitatitve feedback from our domain experts. This feedback is confined to the 24 representative clusters just described. Some important points are as follows:

The approach is promising. Both the SkyServer person and the independent astronomer have confirmed this. The results might not only be useful for the data owner, but for users as well: They help to explore the database, i.e., which combinations of attributes/attribute ranges obviously are important (e.g., Cluster 5). They also offer orientation in the sense 'Which parts of the data do others deem important?'.

Most results are plausible. Cluster boundaries tend to map to meaningful concepts of astronomy. For instance, the closer a *dec*-value is to the equator (*dec*-value 0), the more interesting the object is for an astronomer. The cluster in Figure 1(b) reflects this. However, not all cluster-boundary values are fully clear. Again in Figure 1(b), we do not know yet why the cluster boundary is $dec = 10$ (and not, say, 8 or 15, which would be just as conceivable). Similarly, we do not have an explanation for the specific boundary values along id-type attributes (Clusters 1–4 for instance). These clusters are numerous and require further investigation. On the other hand, our table does not contain clusters on attributes that the astronomer expects to be queried frequently, such as magnitude.

Result presentation should be improved/refined. Both our experts have interpreted the top row as the 'most frequent access area' (and were puzzled that most queries explicitly refer to *Photoz.objid*), but this is not correct in general. Other attributes may be queried more frequently, but the values in queries are spread more evenly over the range, i.e., there is no cluster. This is even likely, since the number of queries per cluster is relatively small ($179,072$ at best, compared to several million queries). What one can infer from the first row is that the values in that range are more likely to be referred to in queries than just outside of the range. A follow-up is that it would be interesting to know how much denser each cluster is, in contrast to its immediate surroundings. We conclude that we should have explained our results more extensively right away, and that there is further information of interest, such as that density drop.

Our results contain useful hints on how the database could be improved. To illustrate, *zooSpec.dec* is queried rather frequently with value -100, even though it is an angle and can only have -90 as its minimal value. Different steps are conceivable, e.g., a tighter definition of value ranges, or a better documentation.

There still are open questions which might be relevant for future research. For example, the astronomer has pointed out that there might be 'test queries' (i.e., queries that are exploratory in nature) which are numerous and influence the clustering result by much and 'final queries', as he calls it. While there might be only relatively few of them, they are important. Finding ways to differentiate between these categories, possibly based on the metadata available, is future work. There also are points that are minor in

Cluster	Cardinality	Area Coverage	Object Coverage	Access area
1	$179,072$	0.24	0.36	$1,237,657,855,534,432,934 \leq Photoz.objid \leq 1,237,666,210,342,830,434$
2	$121,311$	0.19	0.22	$1,115,887,524,498,139,136 \leq SpecObjAll.specobjid \leq 2,183,177,975,464,224,768$
3	$92,177$	0.22	0.21	$1,345,591,721,622,267,904 \leq galSpecLine.specobjid \leq 2,007,633,797,213,874,176$
4	$90,047$	0.25	0.25	$1,416,192,325,597,030,400 \leq galSpecInfo.specobjid \leq 2,183,213,984,470,034,432$
5	$90,015$	0.19	0.25	$PhotoObjAll.ra \leq 210 \wedge PhotoObjAll.dec \leq 10$
6	$82,196$	0.23	0.24	$1,228,357,946,564,438,016 \leq sppLines.specobjid \leq 2,069,493,422,263,134,208$
7	$23,021$	0.17	0.04	$54 \leq SpecObjAll.ra \leq 115$
8	$23,021$	0.23	0.09	$60 \leq SpecPhotoAll.ra \leq 124$
9	$18,904$	0.03	0.01	$(SpecObjAll.class =$ 'star') $\wedge(51,578 \leq SpecObjAll.mjd \leq 52,178) \wedge (296 \leq SpecObjAll.plate \leq 3,200)$
10	$10,141$	0.26	0.27	$(DBObjects.access =$ 'U'$) \wedge ((DBObjects.type =$ 'V'$) \vee (DBObjects.type =$ 'U'$))$
11	$4,006$	0.24	0.18	$55 \leq emissionLinesPort.ra \leq 141$
12	$3,785$	0.21	0.17	$62 \leq stellarMassPCAWisc.ra \leq 138$
13	$1,622$	0.12	0.11	$AtlasOutline.objid > 1,237,676,243,900,255,188$
14	$1,371$	0.16	0.01	$(2 \leq zooSpec.ra \leq 120) \wedge (30 \leq zooSpec.dec \leq 70)$
15	$1,141$	0.10	0.05	$0 \leq Photoz.z \leq 0.1$
16	$1,102$	0.25	0.17	$(0 \leq galSpecExtra.bptclass \leq 3)$ $\wedge(galSpecExtra.specobjid = galSpecIndx.specObjID)$
17	$1,035$	< 0.001	< 0.001	$(sppLines.gwholemask = 0) \wedge (0 \leq sppLines.gwholeside \leq 50)$ $\wedge(sppLines.specobjid = sppParams.specobjid)$ $\wedge(-0.3 \leq sppParams.fehadop \leq 0.5) \wedge (2 \leq sppParams.loggadop \leq 3)$
18	$48,470$	0.0	0.0	$(10 \leq PhotoObjAll.ra \leq 120) \wedge (-90 \leq PhotoObjAll.dec \leq -50)$
19	$41,599$	0.0	0.0	$3,519,644,828,126,257,152 \leq galSpecLine.specobjid \leq 5,788,299,621,113,984,000$
20	$18,444$	0.0	0.0	$3,519,644,828,126,257,152 \leq galSpecInfo.specobjid \leq 5,788,299,621,113,984,000$
21	$18,043$	0.0	0.0	$4,037,480,726,273,651,712 \leq spplines.specobjid \leq 5,788,299,621,113,984,000$
22	$1,358$	0.0	0.0	$(6 \leq zooSpec.ra \leq 115) \wedge (-100 \leq zooSpec.dec \leq -15)$
23	422	0.0	0.0	$-0.98 \leq Photoz.z \leq -0.1$
24	217	0.0	0.0	$3.0 \leq Photoz.z \leq 6.5$

Table 1: Some aggregated access areas (clusters of queries), extracted from SkyServer query log.

nature, e.g., suggestions for further figures describing the clusters.

6.4 Comparison to [4]

We want to learn whether our distance measure affects the result by much. To do so, we compare our method to OLAPClus [4], which has a proprietary distance function (i.e., based on structure) for measuring the (dis-)similarity of (OLAP) queries. The distance function is also applicable to access areas. However, it requires *exact matching* of two atomic predicates and not their overlapping in access areas. Thus, when queries accessing the same data space have very different predicates, e.g., accessing different sets of objects, it is expected that OLAPClus does not group them into the same cluster, i.e., aggregated access areas are lost.

The result of OLAPClus on our data set of SkyServer access areas actually reflects this hypothesis. In particular, OLAPClus produces approximately $100,000$ clusters for Cluster 1 of our method. This is because almost every query in Cluster 1 has its predicate in the form $Photoz.objid = c$ where c is a constant. Similarly, for each of the Clusters 2–4 of our method, OLAPClus outputs about $50,000$ clusters. This does not only cause high redundancy but also loss of knowledge on the interests of users.

The benefits of our method on the other hand are two-fold: (a) succinct output that facilitates post-analysis, and (b) meaningful capture of the access patterns of users.

6.5 Comparison to OLAPClus on Raw Queries

Equation (1) suggests that one could actually obtain aggregated access areas by computing our distance function on the raw queries and clustering them accordingly, without using our extraction method. Essentially this can be done by applying OLAPClus. However, for fair comparison, we replace the exact matching of atomic predicates in OLAPClus by our d_{conj} (see Section 5).

The results show that this version of OLAPClus breaks Clusters 2, 5, 8, 9, 11, 12, 18, 19, 20, and 22 in Table 1. This is because these clusters contain queries of the forms in Section 4.3 and we have proved that directly using predicates as-is may lead to misleading access areas. In addition, the modified OLAPClus yields clusters that do not permit an easy construction of aggregated access areas, as heterogeneous Boolean expressions, resulted from keeping predicates as-is, are put in the same cluster.

6.6 Comparison to Re-querying

Next, we compare our method against the approach that re-issues queries for collecting statistics. Here, we use two performance metrics: efficiency (runtime) and quality of access areas.

Efficiency. Our method processes $100,000$ queries in about 45 seconds on our test machine (Intel® i5-750 CPU with 8GB RAM). There are however queries where the extraction takes rather long time, and in very rare cases, the extraction could not be done within reasonable time (in the range of hours). Looking closer, we find that query execution times of each single step (Parsing, Extraction, CNF, Consolidation) varies between: (a) Parsing: <1 millisecond and 94 milliseconds, (b) Extraction: <1 millisecond and 1333 milliseconds, (c) CNF: <1 millisecond and *undefined*, and (d) Consolidation: <1 millisecond and 95 milliseconds.

The CNF converter, which we took from an open source project, is definitely the weakest point with respect to efficiency. In particular, we discover that the necessary system resources (CPU time, RAM) grow exponentially with the number of predicates the access area currently processed includes. Fortunately, in our total of more than 12 million queries, there are only 471 queries with more than 35 predicates. Such a query often poses a performance bottleneck. To alleviate the issue, we provide a method within our implementation that only considers the first 35 predicates of any query. With this workaround, CNF conversion never lasts longer than 1 hour. More sophisticated processing is left for future work.

Re-issuing queries is far more expensive than our method. In particular, our method is orders of magnitude faster than this naïve approach. Since re-issuing a large number of queries against the SkyServer database does not terminate within a reasonable amount of time, we re-run the queries instead on a sample of the database. The result of this variant is discussed next.

Quality. Compared to re-issuing queries, our method of extracting access areas from the query log provides two advantages. First, we discover empty spaces that are accessed by many queries. As expected, extracting access areas from actual results of queries only yields the areas covered by the database content, i.e., Clusters 18–24 discovered by our method are missed by this approach. Second, our method is able to extract access areas from $1,220,358$ queries that cause errors when being issued to the SkyServer database. Further, our method even extracts access areas of queries that are not written in correct MSSQL code (which is necessary for SkyServer). Such queries often are written in a MySQL dialect such as *SELECT Galaxies.objid FROM Galaxies LIMIT 10*.

The approach that re-issues queries in turn neither yields empty spaces nor is able to process queries with execution errors. All in all, our method offers a more flexible solution towards extracting the access patterns of users from query logs.

7. CONCLUSIONS

Extracting access patterns of database users, i.e., access areas of queries, from query logs is crucial to learn the database usage. This has many applications; one is that it allows to make explicit the research focus of the respective scientific community. However, the task is challenging due to the lack of (a) a formal definition of access area, (b) a mapping of queries to their access areas, and (c) a procedure, including a distance function, to aggregate the access areas of a large set of queries.

In this paper, we have a proposed a solution to each of these challenges. First, we have introduced the novel concept of query access area. It allows the extraction of access areas independent of the database content. Second, we provide a mapping of our notion to all query types occurring in the log, i.e., we enable extraction of access areas in practice. Third, we exploit query overlap for the detection of aggregated access areas that abstract from the individual queries. Domain experts deem our approach interesting. Our case study on the SkyServer query log further shows that our method discovers clusters of access areas that occupy a small fraction of the database content. Some access areas even span empty parts of the data space. Empirical results also show that our method outperforms both a state of the art technique on measuring similarities of queries and an approach that re-issues queries.

In furture work, we plan to experiment with different clustering techniques on our data sets of extracted access areas. Further, we intend to test our method with different distance functions to unveil other interesting access patterns of SkyServer users.

Acknowledgment

We thank Jordan Raddick from Johns Hopkins University for helpful discussion and comments.

8. REFERENCES

[1] R. Agrawal et al. Context-sensitive ranking. In *SIGMOD Conference*, 2006.
[2] J. Akbarnejad et al. Sql QueRIE recommendations. *PVLDB*, 3(2), 2010.
[3] J. Aligon et al. Mining preferences from OLAP query logs for proactive personalization. In *ADBIS*, 2011.
[4] J. Aligon et al. Similarity measures for OLAP sessions. *Knowl. Inf. Syst.*, 37(2), 2014.
[5] F. Becker. Transforming the SDSS SkyServer SQL query log. Master's thesis, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, 2013.
[6] S. Ceri and G. Gottlob. Translating SQL into relational algebra: Optimization, semantics, and equivalence of sql queries. *IEEE Trans. Softw. Eng.*, 11(4), 1985.
[7] G. Chatzopoulou et al. Query recommendations for interactive database exploration. In *SSDBM*, 2009.
[8] G. Chatzopoulou et al. The QueRIE system for personalized query recommendations. *IEEE Data Eng. Bull.*, 34(2), 2011.
[9] A. Cleve and J.-L. Hainaut. Dynamic analysis of SQL statements for data-intensive applications reverse engineering. In *WCRE*, 2008.
[10] M. Ester et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, 1996.
[11] W. Gatterbauer. Databases will visualize queries too. *PVLDB*, 4(12), 2011.
[12] A. Ghosh et al. Plan selection based on query clustering. In *VLDB*, 2002.
[13] A. Giacometti et al. Recommending multidimensional queries. In *DaWaK*, 2009.
[14] T. Grust et al. True language-level SQL debugging. In *EDBT*, 2011.
[15] Y. Ioannidis. From databases to natural language: The unusual direction. In *NLDB*, 2008.
[16] A. K. Jain et al. Data clustering: A review. *ACM Comput. Surv.*, 31(3), 1999.
[17] K. P. Joshi et al. Warehousing and mining web logs. In *WIDM*, 1999.
[18] N. Koudas et al. Relaxing join and selection queries. In *VLDB*, 2006.
[19] G. Koutrika et al. Explaining structured queries in natural language. In *ICDE*, 2010.
[20] S. Mittal et al. QueRIE: A query recommender system supporting interactive database exploration. In *ICDMW*, 2010.
[21] H. Pirahesh et al. Extensible/rule based query rewrite optimization in starburst. In *SIGMOD Conference*, 1992.
[22] F. Silvestri. Mining query logs: Turning search usage data into knowledge. *Found. Trends Inf. Retr.*, 4(1-2), 2010.
[23] V. Singh et al. SkyServer traffic report - the first five years. *CoRR*, abs/cs/0701173:190–204, 2007.
[24] X. Yang et al. Recommending join queries via query log analysis. In *ICDE*, 2009.
[25] Q. Yao et al. Finding and analyzing database user sessions. In *DASFAA*, 2005.
[26] J. Zhang. *Data use and access behavior in escience—exploring data practices in the new data-intensive science paradigm*. PhD thesis, Drexel University, Philadelphia, PA, USA, 2011.

Insights on a Scalable and Dynamic Traffic Management System

Nikolas Zygouras
Department of Informatics and Telecommunications
University of Athens
Greece
nzygouras@di.uoa.gr

Nikos Zacheilas
Department Informatics
Athens University of Economics and Business
Greece
zacheilas@aueb.gr

Vana Kalogeraki
Department Informatics
Athens University of Economics and Business
Greece
vana@aueb.gr

Dermot Kinane
Dublin City Council
Ireland
dermot.kinane@dublincity.ie

Dimitrios Gunopulos
Department of Informatics and Telecommunications
University of Athens
Greece
dg@di.uoa.gr

ABSTRACT

Complex Event Processing (CEP) systems process large streams of data trying to detect events of interest. Traditional CEP systems, such as Esper, lack the required scalability and processing capability to cope with the constantly increasing amount of data that needs to be processed. Furthermore, user defined rules are static so changes in the monitored environment cannot be easily detected. In this paper we investigate the development of a scalable and dynamic traffic management system. Our work makes several contributions: We propose a novel system that combines Esper with a stream processing framework, Storm, in order to parallelize the processing of larger amounts of data. We propose a novel rules' assignment algorithm for distributing Esper rules to the available CEP engines, in a way that maximizes the overall system's throughput. Finally, our system adapts to changes of the environment by processing historical data via Hadoop and dynamically updating the Esper rules based on the generated results. Our work has been evaluated using real data, in several traffic monitoring scenarios for the city of Dublin. Our detailed experimental results indicate the benefits in the working of our approach and the significant increase in the system's throughput when a large number of Esper rules were examined concurrently.

Categories and Subject Descriptors

H.2.4 [**Information Systems**]: Systems—*complex event processing, stream processing*

1. INTRODUCTION

Today there is a large increase in the amount of data that needs to be analysed and processed in real-time in a wide variety of domains, ranging from financial processing [13] to traffic monitoring [8] to healthcare infrastructures [23]. Complex Event Processing systems (CEP) have emerged as a valid solution for analyzing this huge stream of information and detecting events of interest. In a CEP system, user-defined rules process *primitive* events received from a monitored environment in order to detect *composite* phenomena by composing primitive or other composite events using a set of event composition operators. Complex event processing for such Big Data applications is challenging as they need to be able to process high volumes of stream data at low processing latencies.

Traditional CEP systems such as Esper [14] are unable to cope with the current data deluge, mainly because they are based on centralized architectures where the CEP engine receives and processes all incoming events in a single host. Due to this reason, there is a shift in performing the CEP processing in Distributed Stream Processing Systems (DSPS) such as Storm [27], Streams [9] or Spark [3] which are now considered major platforms for data analysis. While such systems provide scalability and fast processing, they lack expressiveness, as the user must provide the actual implementation of the rules that the system has to execute, often needing to express complex tasks and requiring a large amount of code. In contrast, systems like Esper, provide an SQL-like language, EPL (Event Processing Language) for expressing the rules, making it much easier to use and learn. Given the popularity of frameworks such as Storm, Streams and Spark, optimizing the performance of CEP processing in DSPS systems is important, as they do not only provide robust, scalable and reliable solutions to processing fast larger amounts of data, but can also be cost-effective solutions as they can reduce the money paid by users to hosting environments such as clouds. Our architecture uses these frameworks in order to meliorate the value offered from the Big Data systems, scaling the system's *velocity* and *volume*. The system's *value* is optimized as our system is able to support and execute more rules and process larger volume of tuples.

One challenge with rules running in current CEP systems is that they tend to be rather static; this means that their behaviour does not change radically over time. For example, in a traffic management system we may want to be able to detect when a bus is delayed. In most cases this is accomplished with a rule that compares the computed delay for a newly arrived bus trace with a static threshold;

10.5441/002/edbt.2015.65

when this threshold is exceeded an event is triggered [5], [6]. However, using a pre-defined threshold at all times is not beneficial, as the behaviour of the traffic conditions typically change during the course of the day. So, it is of great interest to automatically decide these rules' thresholds. Towards this goal was the work of [25], however it requires a time consuming training phase before it generates the rules that will be used by their CEP system. In contrast, our aim is to dynamically compute the rules' thresholds and change the rules accordingly, in real-time. In our previous work [6] we have illustrated that the Lambda architecture, operating in both batch and stream processing modes, is a promising approach for processing heterogeneous data streams for intelligent urban traffic management. In this work we focus on the **scalability** aspect of our system and illustrate that our system can effectively support multiple concurrently running Esper engines and can **dynamically** adapt to rules' thresholds changes in real-time.

In this work we investigate the development of a CEP system for scalable and dynamic traffic management, that is both powerful and easy-to-use. We provide a system that offers: (i) scalability, (ii) low-latency processing, (iii) ease of use, and (iv) dynamic rule updating to changing system conditions. We focus on the ease of usage because we want the rules running in our system to be easily understood even by non-expert users. Our proposal combines the two approaches (CEP systems and DSPS), exploiting the expressiveness and ease of use of a traditional centralized CEP system such as Esper, and the scalability and fast processing offered by Storm. Supporting dynamic rules is important because it offloads effort from the user as she no longer needs to manually tune the rules' thresholds. Our approach makes the following contributions:

1. We present a novel system architecture that combines Storm, Esper and Hadoop [17], offering a truly scalable and easy-to-use framework for efficient complex event processing. Storm enables us to use a number of Esper engines in parallel, increasing the overall system throughput. Furthermore, by using more engines we are able to concurrently execute multiple Esper rules, further improving the system's performance.
2. We provide algorithms for distributing the Esper rules to the available engines, examining the impact of the assignment on the system's throughput. Our proposed solution aims to balance the load across the engines, so that rules are allocated in a way that all engines process rules with approximately the same amount of input data.
3. We use the Hadoop MapReduce system that is ideally suited for processing historical data. This allows us to compute new thresholds for the running rules and adapt the Esper engines' in real-time for accurate detection of complex events. By using *dynamic* rules we are able to capture the changing conditions of the environment and detect only events of interest.
4. We have implemented our approach and evaluated it using a real traffic monitoring application in the city of Dublin[1]. Our experimental results indicate that our approach is truly scalable, achieves a significant increase in the system's throughput, and can support several concurrently running Esper rules.

2. BASIC COMPONENTS - BACKGROUND

In this section we give an overview of the basic components of our complex event processing framework and discuss some background information and related work.

[1]http://dublinked.com/datastore/datasets/dataset-304.php

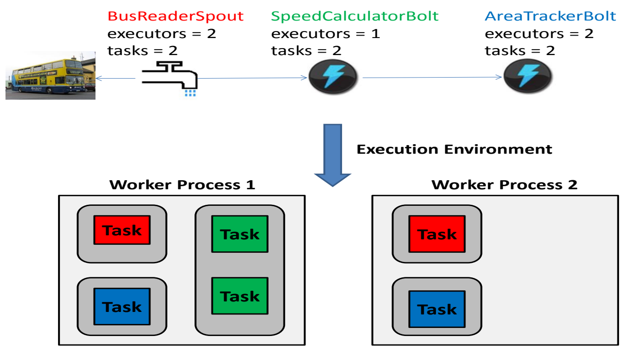


Figure 1: Storm Topology Example

2.1 Basic Framework Components

2.1.1 Storm

Storm [27] has emerged as one of the most commonly used Distributed Stream Processing Engines utilized by major companies such as Twitter [32] and Groupon [16]. It has been successfully employed for processing high volume and intensive workloads in various application domains, where high levels of data throughput and low response latencies are a necessity [26]. Processing massive amounts of data in real-time is achieved by distributing the workload across multiple computers. Applications in Storm are implemented as user-defined topologies. Topologies can be viewed as processing graphs, consisting of nodes that represent user-defined processing operations or primitive event nodes, and edges that represent the streaming of the data. The nodes of a topology graph in Storm can be either *spouts* or *bolts*. Spouts represent the input sources which feed the topology with data, while bolts encapsulate the processing logic. For example, in a CEP application, spouts can be seen as the input sources of primitive or complex events, while bolts are the components that process these events and detect more complex ones.

Users implement the code that will be executed by the Storm components and decide the communication patterns. This essentially represents the subscription of bolts to their input sources. Storm gives the users the capability of deciding how many instances of the implemented bolts' and spouts' code to run in the framework by setting two basic parameters: the number of tasks and executors to utilize. By choosing these parameters, we can increase the parallelism of the topology, making it more scalable. The *executors* parameter (as can be seen in Figure 1) can be used to adjust the number of threads that will execute the processing implemented on the spouts and bolts. The actual processing is performed by the components' *tasks*. Tasks are Java objects containing the user-defined code for the components. Ideally there should be one executor for each task. If the number of tasks is greater than the number of executors, tasks assigned to the same executor are executed in a pseudo-parallel way. In Figure 1, we give an example of a possible assignment of tasks to the corresponding executors in a snapshot of the Storm topology we use for monitoring the traffic conditions in the city of Dublin. Because the SpeedCalculatorBolt has two tasks but only one executor, its tasks are assigned to this single executor. All other components exploit fully the parallelism they can use, thus their tasks are assigned to separate executors.

From an architectural perspective, a Storm cluster consists of a set of physical machines, called *Worker* nodes that are responsible for executing the user topologies, and a Master node called *Nimbus* that coordinates the execution of the topologies. Once a topology has been allocated to the Storm cluster, its executors are assigned to a set of Java processes (*worker* processes) running on the available nodes. Each node is configured with a fixed number of *slots* that will represent the maximum number of *worker* processes that can execute in it. The assignment of the *executors* to the available *worker* processes follows a simple round-robin approach.

2.1.2 Esper

Esper [14] is a Complex Event Processing (CEP) system, applied to streaming data, that triggers actions when the incoming data satisfy some predefined rules. Esper libraries are available for the Java language such as Esper, and for .NET such as NEsper. Esper keeps all the required data structured in memory making the processing fast. The core of the Esper system is the Esper engine which consists of a set of standing queries (or rules). The plethora of technologies that have been developed in Big Data community require the data first to be saved and then to be processed. Esper, on the other hand, provides real time Big Data analytics as it is a 'NoDatabase' technology meaning that no data has to be saved. Esper stores rules in the Esper engine and when new data arrive checks whether or not these rules are fired. This procedure is continuous, as new arriving data are processed serially and the Esper engine responds in real time if any of the stored events meets the constraints. The triggered events can be pushed further into the Esper engine feeding other rules or sent to their listeners. Listeners are associated with rules and define the actions to be taken when the rule is activated. The user can create queries and add them into the Esper engine. These queries are written in Event Processing Language (EPL) and their syntax is similar to *SQL* queries, with *SELECT, FROM, WHERE, GROUP BY, HAVING* and *ORDER BY* clauses. EPL was designed aiming to be similar to the SQL query language. The main difference between EPL and SQL is that EPL uses views instead of tables. Views are the different operations applied to the incoming data to structure data in an event stream. An example of such operation is the expiry policy for events that specify for how long an event will remain in the event stream. Finally, each EPL query defines a sliding or batch window of the incoming stream that it monitors.

2.1.3 Hadoop

The MapReduce programming and execution model [12], along with its open-source implementation Hadoop [17], has emerged as one of the most widely adopted programming models for processing massive-scale datasets. Hadoop has been utilized in a wide variety of application domains including traffic monitoring, stock-market data analysis and financial trading applications. For traffic monitoring applications, such as the one we study, Hadoop can be used to process and analyze historical data in order to compute and store statistics on the stream data, such as to identify normal traffic conditions in different city areas during the course of the day. In the MapReduce model, each computation, or *job*, is modelled as a sequence of two basic operators: *map* and *reduce*. Jobs are automatically parallelized and executed on the available cluster nodes, these are executed as multiple *map* and *reduce* tasks. The map and reduce tasks have the following specifications:

$map(k_1, v_1) \Rightarrow [k_2, v_2]$

$reduce(k_2, [v_2]) \Rightarrow [k_3, v_3]$

In Hadoop, each *map* task is responsible for processing a distinct chunk of the data stored in its distributed filesystem (HDFS [18]). The output of the *map* phase is partitioned using a hash function into a user-defined number of *reduce* tasks. *Reduce* tasks receive their corresponding input data and invoke the *reduce* method on them. All intermediate data generated by the *map* tasks as well as the final results are stored in HDFS for fault-tolerance, but at the cost of extra processing [29], [33].

2.2 Related Work

Stream processing frameworks such as IBM's Infosphere streams [8], Storm [27] and TUD-Streams [9] have been successfully applied for complex event processing. However they lack an expressive language such the one offered by Esper. So the user is responsible to implement all the components required for detecting complex events. Another recently proposed stream processing engine is Spark [36]. Spark aims to unite the worlds of batch and stream processing offering a common framework for both types of computation. Despite its rise in popularity, it is still limited with respect to the expressiveness of the computations, requiring the user to manually implement multiple processing components.

Authors in [2] have focused on the placement of *worker* processors and *executors* of a Storm topology on the available cluster nodes, with the main goal being the minimization of the inter-node communication. Similar scheduling techniques have been proposed in [21], and [34]. These works are orthogonal to ours and can further enhance the working of Storm, increasing the overall system's performance. In [35], the impact of intra-node communication was examined, and it was illustrated that in order to minimize the intra-process communication overhead the number of *worker* processors should be equal to the number of cluster nodes. We adopted this scheduling policy in our framework to minimize the impact of the intra-node communication in the system's performance.

Traffic monitoring has been a field of great interest in the complex event and stream processing community [8], [28]. However, these works detect events based on statically defined *rules* so any updates to the traffic conditions overtime are not taken into account. In contrast, our proposal computes new thresholds for the *rules* and dynamically updates them. Linear Road benchmark [4] is one of the most commonly used platforms for the evaluation of complex event processing frameworks. Many works such as [10] use this framework for their evaluation. However a real dataset such the one we use, can offer a wider variety of events to detect. A recent city transportation application was proposed in [24]. They implemented an application that enables the sharing of taxi rides in a large city, in a way that is beneficial for both citizens and taxi drivers.

With respect to parallelizing the execution of complex event processing systems, work was mainly done by [7]. They increase the parallelism of a complex event procedure by executing multiple instances of the corresponding Finite State Machine, but each with different proportion of the input data. Their approach differs from ours in the fact that their proposal is limited to sequential *rules* while ours can be applied to all types of Esper *rules*. Our system, from an architectural perspective, is similar to the work proposed in [15]. In their proposal they combine Streams [9] with Esper trying to detect events in a football match. However they use only one Esper engine so they do not exploit the parallelism of the DSPS to improve their system's performance.

Authors in [20] propose a system that supports scalable CEP by using more engines and a rules allocation schema that tries to assign rules to engines based on the similarity of their attributes. Our approach differs in two aspects: (1) our framework can scale-up automatically via the features provided by Storm, while (2) our proposed rules allocation algorithm takes into account both the attributes' similarity as well as the expected input rate. In [1], they tackle the problem of distributing the processing of primitive events on the event sources by generating multi-step event acquisition and processing plans with the goal to minimize the event transmissions cost. Also in [31], they propose Next CEP, a distributed complex event processing system that optimizes the usage of the available system resources. This work was evaluated on a fraud detection application, applying their rules in streams with credit card transactions. Finally, in [30], a scalable CEP system was proposed that targets industrial infrastructures, specifically, e-energy applications in the cloud. However, they do not provide any rules allocation algorithm for the distribution of the rules to the nodes.

3. TRAFFIC MONITORING

3.1 Setting Up the Problem

This work is focused on developing a novel architecture and techniques for a scalable and dynamic traffic management system. Our objective is to recognize in real time abnormal traffic events, like accidents and traffic congestions in the Dublin City[2]. The Dublin City traffic control receives data from various voluminous sources, including cameras CCTV, static sensors that measure the traffic flow on several junctions and buses that move in Dublin city. It is not possible for humans to monitor this large amount of data, so the development of a system that receives raw data, processes them and alerts automatically in case of an emergency, is required. An example of an emergency situation in Dublin is presented in Figure 2.

Dublin City Council (DCC) Intelligent Transportation Systems department provided us with the main requirements of a traffic monitoring system applied in Dublin city. These requirements are summarized bellow:

- Their main aim is to be able to identify the spatial locations where the traffic behavior from the buses, obtained through streaming, exceeds the expected normal behaviour for that particular location. An example of this is a rule that checks if in three consecutive bus stops, buses traversing them, reported simultaneously delays greater than the expected.
- Another requirement of DCC is to determine the normal behaviour for different spatial locations. In traffic monitoring systems the traffic behaviour varies for different areas of the city. This behaviour also varies for different hours of day and between weekdays and weekends. It is usual to have greater delays and lower speed in the city centre than the suburbs and greater delays in working hours than the weekends. So it is essential to set up different thresholds which will enable us to model normal traffic behaviour for different spatial locations, hours and days.
- Also it is possible these thresholds to change over time; for example if a new road is constructed the thresholds may be relaxed and the system should adapt to these changes.
- Finally the traffic monitoring system should work in real time, be able to process large amounts of streaming data and respond quickly in unusual conditions.

[2]This work was done in the context of an EU-funded project Insight http://www.insight-ict.eu/

Attribute	Description
Timestamp	the time of the measurement
LineId	the line of the bus
Direction	true or false
GPS position	Longitude and Latitude of the bus
Delay	the seconds that the bus is ahead of schedule
Congestion	true or false
Bus Stop	the id of the closest bus stop
Vehicle Id	The value that is used to distinguish different buses

Table 1: Description of the Dataset

Property	Value
Number of buses	911
Size of data	160 MB per day
Number of lines	67
Data frequency	3 tuples/min per bus
Time interval	6am till 3am

Table 2: Dataset Properties

Figure 2: Traffic accident in Dublin city

The traffic monitoring system that we built has been tested on bus data across Dublin city, provided from DCC. Each bus transmits every 20 seconds information about its position and congestions. The description of the data provided by the buses is given in Table 1. The dataset's properties are described in Table 2. In order to get more meaningful information about the traffic conditions we decided to process further the raw data, enhancing them with new features. For each tuple that the buses transmit, we compute the speed of the bus movement and the change in the delay value from its previously received measurement, labelled as actual delay.

3.2 Our System Architecture

The main goals of our work is to provide a scalable and easy-to-use traffic monitoring system. We propose a system architecture (shown in Figure 3) that consists of: (i) a Distributed Stream Processing System (Storm), (ii) a batch processing framework (Hadoop), (iii) a distributed filesystem (HDFS), (iv) a storage medium (MySQL Server) and (v) multiple Complex Event Processing (Esper) engines used for detecting events of interest. Our architecture bares a lot of similarities with the Lambda-Architecture [22], however we differ in that we exploit the expressiveness of CEP engines to support complex rules. In our framework the queries we execute are user-defined Esper rules, and the merging of the real-time and batch views is done by exploiting the capabilities of Esper (we will discuss this in more detail in Section 4.2).

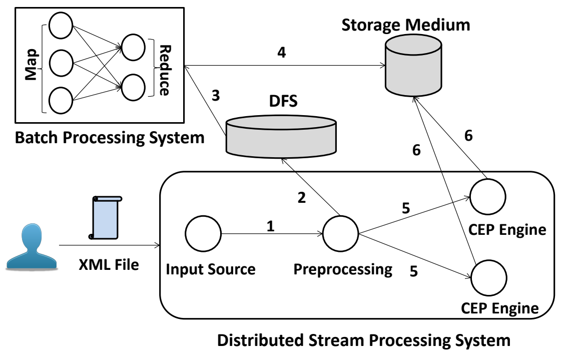


Figure 3: System Architecture

Users in our framework complete an XML file that includes the description of the submitted topology (e.g., spouts, bolts) along with the Esper rules they want to apply to the incoming raw data. We enhanced Storm's library by supporting the creation of topologies via XML. The advantage of this, is that we avoid Storm's standard topology creation procedure, where the user must manually (with Java code) specify how the components should interact with each other. In the simplest case, the user must submit only a spout for specifying the input source along with the rules she wishes to execute. However, for more complex scenarios she can also define extra bolts for pre-processing the raw data before the actual submission to the Esper engines. The pre-processed data before being forwarded to the Esper engines, are stored to a distributed filesystem, HDFS in our case.

Data stored in HDFS are used for computing the threshold values that will be utilized by the Esper rules for detecting complex events. This computation is done by the batch processing layer, in our case Hadoop. We chose Hadoop, since the stored data may increase significantly in size, as new data constantly arrive in our framework. We apply simple MapReduce computations in these historical data and the calculated threshold values are stored to the storage medium in order to be accessible from the Esper engines. In our current implementation the storage medium is a MySQL server but it can easily be substituted by a distributed solution, such as Cassandra [11].

Storm via its scalability features (specifically the usage of more tasks per bolt) enables us to increase the number of Esper engines used for the event detection. This is achieved by increasing the parallelism of the bolt that will be responsible for running the Esper engine. So, if we increase the number of tasks for a bolt, we end up having multiple concurrently running engines. To make full usage of the parallelism, when we increase the number of tasks of the corresponding bolt, we also increase the number of executors in order to run each engine in a separate thread. Furthermore, we allocate the executors into different *worker* processors to make sure that each cluster node will be assigned with the same number of Esper engines.

Using more engines increases the system's throughput and the number of rules that can execute concurrently. Special care must be given on how to allocate the user-defined rules to the available engines (Section 4.2), as we want to fairly distribute the rules, avoiding overloaded situations. Furthermore, rules in our framework should dynamically adapt to new thresholds as threshold values change overtime by the computations performed from the batch processing layer. We examined several techniques (Section 4.3.1) for collecting this information from the storage medium and join them with the streaming data.

3.3 Rules Description

We define rules that allow us to detect events of interest in the traffic data. Finding out where there may be traffic incidents or identifying anomalies in traffic patterns, are examples of events of interest for the Dublin City traffic management system. All events signify certain activities like low speed or increased delays at a particular area. In general the complexity of the rules can vary significantly, as each rule has different characteristics. For example, identifying if a bus is delayed might be simple to detect, while detecting anomalies in traffic patterns might involve computations over multiple simpler events.

We created a generic rule template that checks if the reported attributes from the buses aggregated in different locations, exceed the thresholds. This generic template was selected after discussion with the traffic experts in the Dublin City Council. The rule template has the following parameters: *bus data attribute*, *spatial location* and *window length*. The *bus data attribute* corresponds to the bus data field or fields that the rule checks if they exceed the predefined threshold. Example of these fields are the buses' delay or speed. We monitor different attributes in order to improve our knowledge of the traffic conditions. The *spatial location* is the spatial area that the rule checks for abnormalities. The reason why we selected to create rules for different areas is that traffic jams have spatial extent and in order to identify them we have to search for abnormalities in these areas. The *window length* is the window size of the stream that the rule keeps in memory for processing. In our scenario we compute the average value of all the values in the streaming window in order to compare it with the corresponding thresholds. We used window-based streams because traffic data are very noisy and taking the average value makes them smoother. The generic rule described above has the following format and is fired when the incoming data satisfy the following condition:

$$\wedge f(attribute_i, l, s) > threshold(attribute_i, s)$$

$$threshold(attribute_i, s) = mean(attribute_i, s) + stdv(attribute_i, s)$$

where $attribute_i$ is the *data attribute* that we check, f is the operator that is applied in the stream of data, l is *window length* of the stream that we monitor and s is the *spatial location* in the city map that we monitor. This generic rule's complexity changes overtime as the different *spatial locations* do not have fixed $mean$ and $stdv$.

The EPL code that implements the generic rule template, described above, is presented in Listing 1. The EPL rule contains three streams as it is defined in the $FROM$ clause. The first stream consists of the last event that arrived in the system. The second stream contains the last l values that arrived in the system and have the same $location$ as the last event. The last stream named *thresholdLocation* contains all the thresholds for all the possible locations for different hours of day and for weekdays and weekends. In addition, the streaming data join with this stream in order to retrieve their thresholds. The rule is fired when the average value of a specific $attribute$ is greater than the corresponding threshold, for a specific $location$, $hour$ and day.

Listing 1: Esper EPL Rule template

```
SELECT *
```

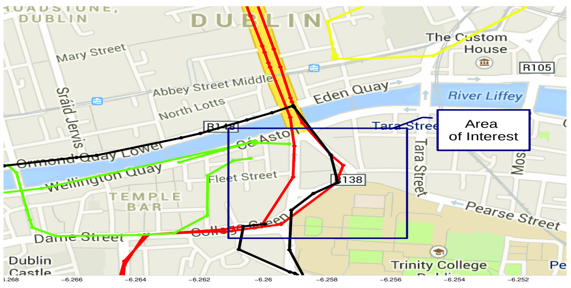


Figure 4: Bus trajectories in a specific area of interest

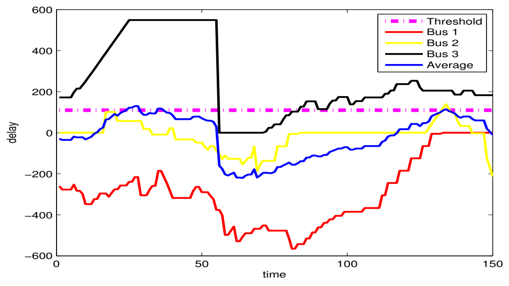


Figure 5: Evolution of delay value for 3 different buses

```
FROM
  bus.std:lastevent() as bd,
  bus.std:groupwin(location).win:length(l) as bd2,
  thresholdLocation.win:keepall() as thresholds
WHERE
  bd.hour=thresholds.hour and
  bd.day=thresholds.day and
  bd.location=thresholds.location and
  bd.location=bd2.location
GROUP BY
  bd2.location
HAVING
  avg(bd2.attribute) > avg(thresholds.attribute)
```

An example of a rule is presented in Figures 4 and 5. Figure 4 shows the trajectories of 3 buses moving in the Dublin city and the corresponding area of interest. The rule checks for abnormalities in this area. The rule is fired when the average delay's value from all the buses that move in the area of interest is greater than the threshold for the particular area. Figure 5 shows the evolution of the 3 buses that are in the bounding box, the average value of delay for the 3 buses and the threshold for this area. The rule is fired when the delay's moving average exceeds the threshold value.

4. DESCRIBING THE COMPUTATION

In order to tackle the traffic monitoring problem we decompose it into three main components:

1. **Off-line Computation.** The computations performed by this component enable us to support *dynamic* rules.
2. **Start-Up Optimization.** This component optimizes the system's parameters according to the user's requirements.
3. **On-line Processing.** The component that is responsible for the actual processing of newly arriving data as well as to detect events of interest.

Our objective in this work is to improve throughput and be able to process as big datasets as possible. We provide algorithms for all three components but we focus on the **Start-Up Optimization** component because this defines the instance of the architecture we run. The first two components are used in order to optimize the performance of the third component.

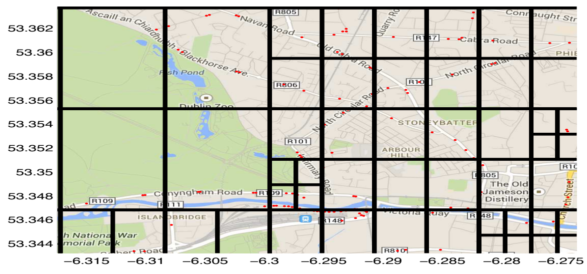


Figure 6: Region Quadtree with bus stops

4.1 Off-line Computation

This component is responsible to prepare the system in order to be initialized and support it in possible future changes. Also it makes our system dynamic as it could change the system's parameters over time and make it to adjust to new, dynamically posed, requirements.

4.1.1 Spatial Indexing

The Dublin City Council requested to be able to monitor city's traffic conditions at different spatial extent, e.g. from small city blocks to whole road lines. For this reason the rules were defined with a hierarchical decomposition regarding the spatial locations. This hierarchical decomposition was not given to us, so we partitioned the map in sub-areas. In order to partition a map there are different approaches that we could apply, including the Region quadtree data-structure, Grids, Voronoi diagrams or even arbitrary shapes that include areas of the city. In our application we utilized the Region quadtree.

The quadtree represents a partition of the space in two dimensions by decomposing each region into four equal sub-regions, and so forth. The region quadtree is created by adding some initial data points to it and then splitting until each region keeps a maximum number of data points. So the resulting tree is not always balanced. In our case the quadtree was created by adding important coordinates of the Dublin city, e.g. main road segments. Because these points are not equally distributed in the city, as can be seen in Figure 6, the regions created by the quadtree are unbalanced. The user decides the spatial extent of the monitoring by specifying in Esper rules, either the layer of the Quadtree she wants to examine or some explicit area of interest.

4.1.2 Bus Stops

Furthermore, we decided to monitor the traffic condition at areas near bus stops in Dublin city. Bus data are noisy, especially when buses report their stops. More particularly, we observed that a specific bus stop is reported at different locations. Also buses reported that they were stopped while they were actually moving. Also nearby bus stops seem to have different ids. We decided to calculate new bus stops and create a tool, that for each line, direction and GPS position, will identify the closest bus stop.

In order to deal with the problem of identifying the bus stops we applied the DENCLUE [19] clustering algorithm in the GPS loca-

Parameters	Values
Output	Rule's Latency
Input 1	length window of a rule, l
Input 2	number of thresholds in the Engine that rule joins with, t

Table 3: Parameters of Single Rule Latency Function

tions, where the buses reported that they just reached the bus stop. Initially DENCLUE added a 2-dimensional Gaussian distribution with $\sigma = 20m$ at each data point and then added all the Gaussians in order to calculate the global density. Then for each data point we identified its local maxima named as density attractor and we kept together all the points that their density attractors were close. These clusters do not consider the bus direction (i.e. a cluster may have bus stops, where buses move in one and the opposite direction). In order to keep the different directions we decided to split the clusters further. Our approach works as follows: We found the average angle that the bus had when it entered in the cluster per line and direction, and then we placed in the same subcluster the bus lines and directions that had similar average angle. Then for each new set of GPS position, line and direction it is possible to find the closest subcluster. For the rest of this paper we will call these subclusters as bus stops.

4.1.3 Supporting Dynamic Rules

Given that we look for abnormalities during the course of the day for different spatial locations, we compute statistics continuously and update the rules accordingly. These statistics are calculated using Hadoop jobs. The jobs are invoked periodically, e.g., every one hour, to compute statistics for the different spatial locations in the upcoming time window, e.g. for the next hour. Specifically, the job calculates the mean and standard deviation of the parameters defined in Table 6 (see Section 5) for the different locations (e.g. quadtree areas or bus stops). In the map phase we retrieve the historical data from HDFS and then emit them to the reduce tasks. The reducers aggregate the parameters' values for the different spatial locations and then compute the mean and the standard deviation. The results are stored in our MySQL server and are retrieved during the online processing to be used as thresholds for the running rules.

4.1.4 Estimate Engine's Latency

A key factor of our analysis is the estimation of each Esper engine's latency. We build a model that takes as parameters the set of rules to run, their characteristics, the number of available cluster nodes and Esper engines, and estimates the latency of each engine. The model's architecture is presented in Figure 7. In order to do this estimation we created three regression functions that are presented bellow:

- **Single Rule Latency Function (Function 1)** This function estimates the latency of a rule that has window l and t thresholds. As we observed that these are the two main components that affect the latency of a rule. The input and the output of this function are presented in Table 3.
- **Multiple Rules Latency Function (Function 2)** The second function estimates the total latency of an Esper engine to process a tuple when we place multiple rules in the same engine. This function takes as input the latency calculated from the first function. If the user inserts rules with different format as

Parameters	Values
Output	Engine's Latency
Input 1	Latency of rule 1
Input 2	Latency of rule 2

Table 4: Parameters of Multiple Rules Latency Function

Parameters	Values
Output	Latency Engine$_i$
Input 1	Latency Engine$_i$
Input 2	Latency Engine$_j$
Input 3	Latency Engine$_k$

Table 5: Engine's Latency Function

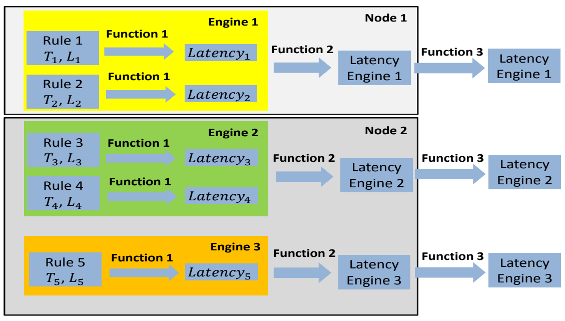


Figure 7: Estimation Model

ours, Single Rules Latency Function is not reliable, thus we calculate the latency of the rule running in a single engine and then insert in the second function this information. If we place more than 2 rules we will call this function sequentially, e.g. the output of this function will be fed again as its input. The input and the output of this function are presented in Table 4.

- **Engine's Latency Function (Function 3)** Finally the last function is used for estimating the latency of an engine's rules if it is placed in the same cluster node with other engines. The latency of processing incoming events increases if a cluster node is overloaded with many engines. The input and the output of this function are presented in Table 5, where three Esper engines run in the same cluster node.

4.2 Start-Up Optimization

This component is responsible for setting up the system, thus, is executed before the actual Storm topology starts processing incoming data. Initially it collects all the rules that the user chose to run and analyses their requirements. In the next step, the component decides how to allocate these rules to different Esper engines. The allocation is done based on the regression model explained in the Section 4.1.4. Furthermore, the component's optimizations can be invoked periodically (or when new rules are submitted to the framework) to adjust the rules allocation to the current system's conditions.

4.2.1 Rules Partitioning

Balancing the data processed by the Esper engines is one of the key components to improving the overall system throughput. The rules

Rule's Partitioning Component

```
Input: $Engines$ the set of Esper engines to use, $rule$ the rule that
needs to be partitioned
$Region\_Rates$ ← Retrieve regions' input rates for layer
$rule.quadtree\_layer$
Sort $Region\_Rates$ in descending order
for all $engine_i$ in $Engines$ do
    $rate[engine_i] = 0$
end for
for all $region$ in $Region\_Rates$ do
    $less\_loaded = engine_1$
    $min\_rate = rate[engine_1]$
    for all $engine_i$ in $Engines$ do
        if $min\_rate > rate[engine_i]$ then
            $min\_rate = rate[engine_i]$
            $less\_loaded = engine_i$
        end if
    end for
    Assign $region$ to $less\_loaded$ engine
    $rate[less\_loaded] = rate[less\_loaded] + region.rate$
end for
```

Algorithm 1: Rule's Partitioning Algorithm

Rules Allocation Component

```
Input: $N$ number of Esper engines, $Groupings\_Set$ set of the
groupings with their corresponding rules
for all $grouping_i$ in $Groupings\_Set$ do
    $scores[grouping_i]$ ← Estimate $grouping_i$'s rules score with 1
    engine
    $engines[grouping_i] = 1$
end for
$N = N - |Groupings\_Set|$
for $j = 1$ to $N$ do
    $max\_score = 0$
    $chosen\_group = grouping_1$
    for all $grouping_i$ in $Groupings\_Set$ do
        $estimated\_score$ ← Estimate $grouping_i$'s rules score for
        $engines[grouping_i] + 1$
        if $max\_score < estimated\_score$ then
            $max\_score = estimated\_score$
            $chosen\_group = grouping_i$
        end if
    end for
    $scores[chosen\_group] = max\_score$
    $engines[chosen\_group] = engines[chosen\_group] + 1$
end for
```

Algorithm 2: Rules Allocation Algorithm

that our system examines detect abnormalities in buses' reported values at different spatial locations. Thus, we partition the rules by sending the data that correspond to different spatial locations into different Esper engines. We follow a partitioning schema that partitions a rule's spatial locations to different engines based on their input rates. The tuples are sent to the appropriate engine according to this schema. We consider as the input rate of a spatial location, the amount of bus traces expected to be processed by the engine in that location. We have some initial knowledge about these rates (e.g. from historical data) and incrementally update them while the application runs. As you can see in Algorithm 1, the rule's examining locations are partitioned in a way that all engines will receive approximately the same aggregated input rate.

4.2.2 Rules Allocation

One of the key components for achieving fast processing of the constantly arriving new primitive events is the allocation of the rules to the available Esper engines. We exploit the hierarchical structure of our rules and provide an efficient assignment of the rules to the engines. Recall that the running rules examine spatial locations. These locations may overlap as one rule may monitor the hole city and other rules some specific roads or neighbourhoods.

Because we have multiple rules per spatial layer and a limited number of engines, we might need to group together rules that belong to different layers. This approach can be beneficial because we avoid data re-transmissions. If we assign each layer in a different engine, newly arriving primitive events must be transmitted to all the engines, limiting the benefits of the Storm's parallelism and adding extra traffic between the cluster nodes. Conversely, if we put all rules examining the second and third quadtree layers in the same grouping, we would not have to sent new events twice as areas of the third layer will be assigned to the same engine with their parents from the second layer. We achieve this by partitioning rules' locations (see 4.2.1) based on the higher possible layer. So in the previous case, we would partition the rules based on the spatial locations belonging to the second layer of our quadtree.

We propose a greedy algorithm (Algorithm 2) that receives as input a possible set of groupings of the different layers examined by the rules, and provides an allocation to the available engines in a way that maximizes a score function. The time required to process the input data for the rules of $grouping_i$ in $Engine_j$ is given by the following formula:

$$time_{i,j} = inputRate_i \times latency_j \tag{1}$$

The score for each grouping will be related to the minimum time required to process its set of tuples in the engines it has been assigned.

$$score_i = \sum_{i=1}^{W_i} w_i \times \min_{j \in Engines}(time_{i,j}) \tag{2}$$

where W_i is the number of rules in $grouping_i$. Also w_i is the weight of a particular rule. The traffic management operator may select that some rules are more important than others. For example, an appropriate policy may be to place higher weights in the rules that take much time to execute.

Our algorithm is not a simple variation of the Bin-Packing problem but the process we follow is a bit more complex. If we place more than one rule in the same engine, the estimation of the engine's latency is not trivial, as it varies for different combinations of rules. For this reason we propose an algorithm that utilizes the regression model for estimating the observed latency when we allocate the rules to multiple engines and then computes the corresponding score via Equation 2. The algorithm first gives in each grouping a separate engine to execute. So if the initial grouping considers the root layer with the second layer together, while the third layer separately, then the algorithm would allocate one engine for the first pair of layers and a second engine for the third layer's rules. Furthermore for each grouping we estimate its achieved score for this initial assignment.

In the next step of the algorithm, we estimate for each grouping its score if we had added an extra engine for that particular grouping. The grouping that leads to the greater score increase is the one that will use the extra engine. This approach enables us to maximize the achievable score without examining all the combinations of layers and engines. In each step we keep the new score estimation for the chosen grouping and increase the number of engines appropriately. We repeat this procedure until all engines have been utilized.

4.3 On-line Processing

This component consists of the Storm topology and the Esper engines. The rules are allocated to these engines via the techniques proposed in the **Start-Up Optimization** component.

4.3.1 Retrieve Batch Generated Data

Esper rules do not have a unique threshold but have different thresholds for different input data (i.e. different bus stop, areas etc). Recall that these values are computed by Hadoop and stored in an SQL server. These thresholds are retrieved using the SQL query that is presented in Listing 2. This query takes as parameter the value s which tunes the value of the threshold as s times the standard deviation away from the mean.

Listing 2: SQL Query that retrieves the thresholds

```
SELECT DISTINCT
  attr_mean+s*attr_stdv as thresholdLocation,
  currentHour,
  dateType,
  areaId1
FROM
  statistics_attribute
```

In order to feed the rules with these thresholds we tested the following three methods:

- **Join with Database** Each new tuple that arrives in an Esper engine will do a join with the database in order to retrieve the corresponding threshold.
- **Create Multiple Rules** Retrieve all the thresholds from the database in advance and for each possible combination create the relevant rule.
- **Add the Thresholds in an Esper stream** Retrieve all the thresholds from the database and add them in a new Esper stream. Each new tuple will join with this thresholds' stream.

4.3.2 Storm Topology

In Figure 8, we saw the Storm topology that we created for the described traffic monitoring application. The application consists of seven components. Newly arrived bus traces are emitted to the topology from the BusReader spout. In our current implementation the traces are stored in csv files so we use this spout for reading the stored data. The PreProcess Bolt is responsible for adding extra information such as the vehicle's speed and direction. These data are forwarded to the Area Tracker bolt which detects the areas in the different quadtree layers where the vehicle currently resides. Each task of this bolt has an instance of the Region Quadtree and queries it to find the areas that the new trace belongs. The BusStops Tracker bolt also adds a new attribute in the examined trace, specifically it adds the bus stop id. Then the data are emmitted to the Splitter Bolt that is responsible to send each tuple to the appropriate Esper Bolt task. As we mentioned above our system consists of many Esper engines located in different tasks of the Bolt. It is crucial to route each bus data tuple to the appropriate Esper engine as each engine examines different spatial locations (Section 4.2.1). Finally the Esper Bolt executes the user-defined *rules* that are used for detecting unusual traffic conditions in the city. We have multiple tasks of this bolt to exploit the inherent parallelism of Storm. Detected events by the EsperBolt are forwarded to the EventsStorer bolt which stores them to a pre-decided storage medium, in our case a MySQL server.

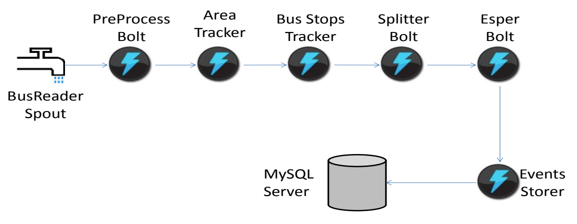


Figure 8: Traffic Monitoring Topology

Parameters	Values
Attribute:	Delay, Actual Delay, Speed, Delay and Congestion, All
Location:	Bus Stops and Quadtree Areas
Window Length:	1, 10, 100, 1000

Table 6: Parameters of the generic rule template and the corresponding values

5. EVALUATION

We have performed an extensive experimental study of our approach on our local cluster consisting of 7 VMs, running on three actual nodes. Each VM had attached one CPU processor and 2 GB RAM. All VMs were connected to the same LAN and their clocks were synchronized with the NTP protocol. We used Storm 0.8.2, Hadoop 1.2.1 and Esper 5.1. We used a separate node in our cluster where the Storm Master process (Nimbus) executed to avoid overloading one of the VMs. In this work, we use the values in Table 6 as parameters of the generic rule template described in Section 3.3. For our evaluation we fed our system with bus traces from the period of 1st to 31st January (4 GB in total) at full speed, so without any delay between the tuples inter-arrivals. We followed this policy to stretch our system's performance, specifically every second our application received $60,000$ bus traces.

We focused on the performance of the bolt that runs the Esper engines, as it is responsible for the more heavy-weight processing and also because it is the part of the topology where our optimizations were applied. Two main **metrics** were considered:

1. The achieved *throughput* in regards to the number of input data that are processed in a fragment of 40 seconds
2. The average *latency* to process a single input tuple. Again we considered a time period of 40 seconds.

To collect these data we enhanced Storm with an extra monitor thread per *worker* processor, that periodically (every 40 seconds in our case) reports these metrics for each bolt's task to the Nimbus node. The Nimbus aggregates these data to compute the final monitor metrics per bolt.

5.1 Regression Evaluation

In order to build the regression model, described in Section 4.1.4, that estimates the latency for an engine if we add in it two sets of rules, we used polynomial regression. Initially we ran several experiments in order to build the appropriate dataset and we splitted it in training and test set. Then we feeded a first and a second order polynomial regression model in this dataset. From the experiments

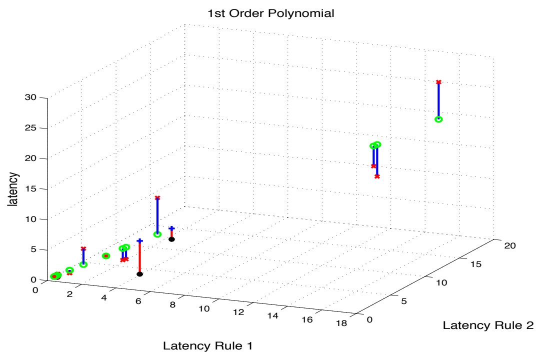
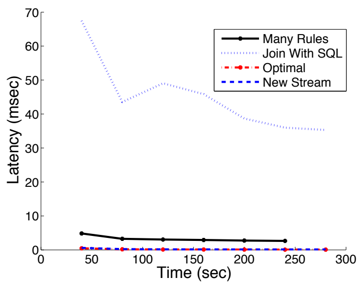
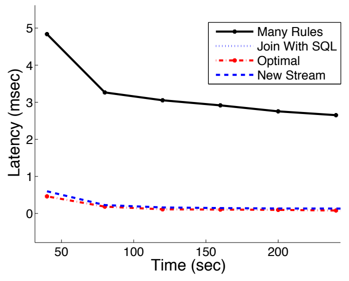


Figure 9: Polynomial for Multiple Rules Latency Function



(a) Observed latency for different retrieval methods



(b) Zoomed observed latency for different retrieval methods

Figure 10: Performance of retrieving location thresholds methods

we identified that the first order polynomial regression has less average absolute error (around 60%) than the second order. This is the reason why we selected to use this model in order to model this function. The identified function is: $0.0077598 \times latency_1 + 2.3016 \times e^{-05} \times latency_2 + 2.4717$. The generated model is presented in Figure 9. We followed a similar approach for creating the other two regression functions.

5.2 Retrieving Results From Storage Medium

We evaluated our three proposed techniques for dynamically retrieving the rules' thresholds from the storage medium, depicting their impact on the observed latency. We also provide results when a static threshold is used. In this case no data needs to be retrieved from the storage medium. This depicts the optimal scenario where we do not have the retrieval overhead. As you can see in Figures 10(a), 10(b), using an inner SQL query for each rule deteriorates the performance of the framework because for each incoming tuple we have to join the tuples' attributes with the ones stored in the storage medium, leading to a significant increase in the latency.

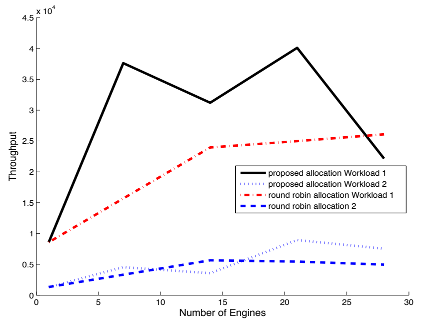


Figure 11: Rules Allocation Throughput Performance

In regards to the two other methods you can see that using a new stream for the thresholds (blue line in the Figures) has latency comparable to the one observed in the no threshold scenario. The multiple rules approach leads to an increased latency because the engines are overloaded with multiple rules, so there is a significant difference compared to the new stream approach as you can see more clearly in Figure 10(b). For the remaining experimental evaluation we used the new stream approach.

5.3 Rules Partitioning

In regards to partitioning the rules to their allocated engines, we compared our proposal with two other approaches:

1. **All Grouping**: Rules' spatial locations are partitioned to the engines similarly to our approach, but newly arrived tuples are emitted to every engine.
2. **All Rules**: All engines have all the rules' locations allocated to them and each incoming tuple is forwarded to the engine that was decided by our partitioning schema.

We compared them with our algorithm when 10 rules are running. Five of the rules examined each of the different attributes (see Table 6) for the bus stops, while the other five monitored the leaves of the quadtree. All rules had 100 tuples in their window length. As you can see in Figures 12, 13, our partitioning proposal achieves a larger increase in the system's throughput, because the system is not overloaded with extra tuples (as the **All Grouping** technique does) and also the engines are not overloaded with rules as in the **All Rules** scenario.

5.4 Rules Allocation

We evaluated the performance of our proposed allocation algorithm, when rules belonging to different quadtree layers must be allocated to the available engines. We used two Workloads based on the attributes and locations described in Table 6. Workload 1 used 1, 10 and 100 window lengths while in Workload 2, rules had 100 and 1000 as window lengths. We compared our algorithm with a simple round-robin approach that considers the rules based on the layer of the quadtree they belong. The algorithm assigns the engines to these layers via a round-robin fashion. As you can see in Figure 11 our algorithm achieves better results because it allocates rules from different layers together avoiding the overhead of retransmissions that occur when the round-robin algorithm is applied. Specifically our algorithm allocated for both workloads all rules together, until fourteen engines were considered. When fourteen engines were used, rules concerning the bus stops were

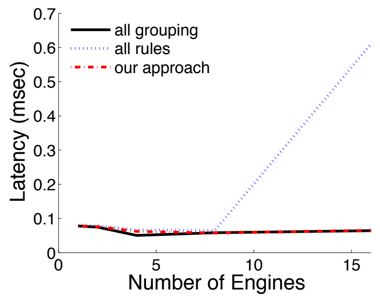


Figure 12: Observed latency for different partitioning approaches

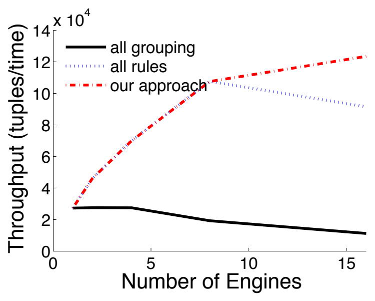


Figure 13: Achieved throughput for different partitioning approaches

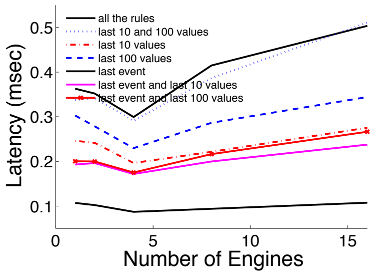


Figure 14: Observed latency for different workloads

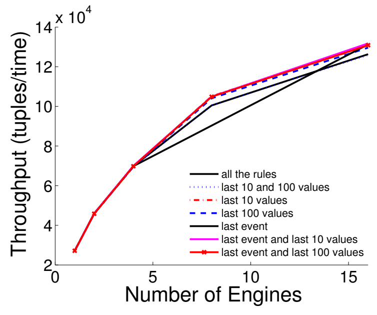


Figure 15: Achieved throughput for different workloads

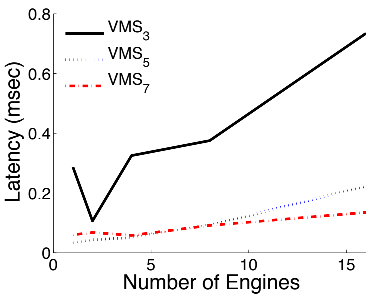


Figure 16: Observed latency for different number of VMs

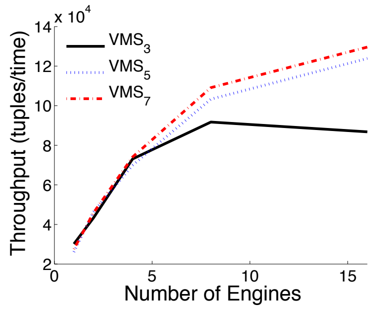


Figure 17: Achieved throughput for different number of VMs

examined in their own dedicated set of engines, and all the other rules were put together to the remaining engines. So our approach maximizes the overlapping between the layers and thus minimizes the required data re-transmissions.

5.5 Different Workloads

We also examined the performance of our system when different workloads are assigned to the available Esper engines. In these experiments we used our proposed allocation algorithm, and examined how our system scales with different workloads. We considered three different workloads based on the rules running in our application. In our evaluation we also examined cases where these workloads were issued concurrently.

- **Last Event**. Consisted of ten rules that kept only one previous tuple in their time window. Five rules examined the attributes values at the bus stops, while the other five monitored the attributes in the leaves layer of the quadtree.
- **Last Ten Values**. Consisted of ten rules that kept ten previous tuples in their time window. Rules had the same attributes and locations as the **Last Event** workload.
- **Last One Hundred Values**. Similarly, this workload consisted of ten rules with the same structure as the other two workloads, only this time we kept one hundred previous tuples in the rules' time windows.

As you can see in Figure 15, our system is able to achieve a steady increase in the overall system's throughput even when we apply all the workloads at the same time.

5.6 Scalability

Finally we evaluated the scalability of our framework when we vary the number of VMs that were used. We examined the framework's performance for 3, 5 and 7 VMs. We used the last workload from the previous section for these experiments. In Figures 16, 17, you can see that when more VMs are available we achieve a steady throughput increase. In regards to latency we can see how overloading the system can lead to its significant increase. For example in the 3 VMs case, using more than four Esper engines leads to a huge increase in the observed latency. Also you can observe that the best results in regards to latency occur when we do not exceed the available processing resources (CPU cores).

6. CONCLUSION

In this paper we presented a novel traffic management system for detecting complex events in the city of Dublin. We proposed a new system architecture that combines Storm, Esper and Hadoop, offering a truly scalable and easy-to-use framework for efficient complex event processing. We provided algorithms for allocating the rules to the available Esper engines and processing historical data in order to be able to support *dynamic* rules. Our experimental results in our local cluster indicate a clear improvement in the system's performance.

7. ACKNOWLEDGMENTS

This research has been co-financed by the European Union (European Social Fund ESF) and Greek national funds through the Operational Program Education and Lifelong Learning of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thalis-DISFER, Aristeia-MMD, Aristeia-INCEPTION Investing in knowledge society through the European Social Fund, the FP7 INSIGHT project and the ERC IDEAS NGHCS project.

8. REFERENCES

[1] M. Akdere, U. Çetintemel, and N. Tatbul. Plan-based Complex Event Detection across Distributed Sources. *Proceedings of the VLDB Endowment VLDB Endowment Hompage archive Volume 1 Issue 1, Pages 66-77, August*, 2008.

[2] L. Aniello, R. Baldoni, and L. Querzoni. Adaptive Online Scheduling in Storm. *DEBS*, 2013.

[3] Apache's Spark. `https://spark.apache.org`.

[4] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear Road: A Stream Data Management Benchmark. *VLDB '04 Proceedings of the Thirtieth international conference on Very large data bases*, 2004.

[5] A. Artikis, M. Weidlich, A. Gal, V. Kalogeraki, and D. Gunopulos. Self-Adaptive Event Recognition for Intelligent Transport Management. *IEEE Conference on Big Data, 319-325*, 2013.

[6] A. Artikis, M. Weidlich, F. Schnitzler, I. Boutsis, T. Liebig, N. Piatkowski, C. Bockermann, K. Morik, V. Kalogeraki, J. Marecek, A. Gal, S. Mannor, D. Kinane, and D. Gunopulos. Heterogeneous Stream Processing and Crowdsourcing for Urban Traffic Management. *in Proc. 17th International Conference on Extending Database Technology (EDBT), Athens, Greece, March 24-28, pp. 712-723*, 2014.

[7] C. Balkasen, N. Dindar, M. Wetter, and N. Tatbul. RIP: Run-based Intra-query Parallelism for Scalable Complex Event Processing. *DEBS*, 2013.

[8] A. Biem, E. Bouillet, H. Feng, A. Ranganathan, A. Riabov, O. Verscheure, H. Koutsopoulos, and C. Moran. IBM Infosphere Streams for Scalable, Real-time, Intelligent Transportation Services. *SIGMOD '10 Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010.

[9] C. Bockermann and H. Blom. The streams framework. *Technical Report 5, TU Dortmund University*, 2012.

[10] I. Botan, P. M. Fischer, D. Kossmann, and N. Tatbul. Transactional Stream Processing. *EDBT '12 Proceedings of the 15th International Conference on Extending Database Technology*, 2012.

[11] Cassandra. `https:/cassandra.apache.org`.

[12] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *OSDI*, 2004.

[13] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Towards Expressive Publish/Subscribe Systems. *EDBT'06 Proceedings of the 10th international conference on Advances in Database Technology Pages 627-644*, 2006.

[14] Esper. `esper.codehaus.org`.

[15] A. Gal, S. Keren, M. Sondak, M. Weidlich, H. Blom, and C. Bockermann. Grand Challenge: The TechniBall System. *DEBS '13 Proceedings of the 7th ACM international conference on Distributed event-based systems Pages 319-324*, 2013.

[16] Groupon. `http://www.groupon.com`.

[17] Hadoop. `http://lucene.apache.org/hadoop`.

[18] HDFS. `hadoop.apache.org/docs/r1.2.1/hdfs_design.html`.

[19] A. Hinneburg and D. A. Keim. An Efficient Approach to Clustering in Large Multimedia Databases with Noise. *KDD*, 1998.

[20] K. Isoyama, Y. Kobayashi, T. Sato, K. Kida, M. Yoshida, and H. Tagato. Short Paper: A Scalable Complex Event Processing. *DEBS '12 Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems Pages 123-126*, 2012.

[21] V. Kravtsov, P. Bar, D. Carmeli, A. Schuster, and M. Swain. A scheduling framework for large-scale, parallel, and topology-aware applications. *Journal of Parallel and Distributed Computing, Volume 70, Issue 9, Pages 983 - 992*, September 2010.

[22] Lambda Architecture. `lambda-architecture.net`.

[23] M. Liu, M. Ray, D. Zhang, E. A. Rundensteiner, D. J. Dougherty, C. Gupta, S. Wang, and I. Ari. Realtime Healthcare Services Via Nested Complex Event Processing Technology. *EDBT '12 Proceedings of the 15th International Conference on Extending Database Technology Pages 622-625*, 2012.

[24] S. Ma, Y. Zheng, and O. Wolfson. T-Share: A Large-Scale Dynamic Taxi Ridesharing Service. *ICDE*, 2013.

[25] A. Margara, G. Cugola, and G. Tamburrelli. Learning From the Past: Automated Rule Generation for Complex Event Processing. *DEBS*, 2011.

[26] R. McCreadie, C. Macdonald, I. Ounis, M. Osborne, and S. Petrovic. Scalable Distributed Event Detection for Twitter. *BigData Conference: 543-549*, 2013.

[27] Nathan Marz's Storm. `https://github.com/nathanmarz/storm`.

[28] K. Patroumpas and T. Sellis. Event Processing and Real-time Monitoring over Streaming Traffic Data. *Web and Wireless Geographical Information Systems Lecture Notes in Computer Science Volume 7236, pp 116-133*, 2012.

[29] S. Schelter, S. Ewen, K. Tzoumas, and V. Markl. All Roads Lead to Rome: Optimistic Recovery for Distributed Iterative Data Processing. *CIKM*, 2013.

[30] B. Schilling, B. Koldehofe, U. Pletat, and K. Rothermel. Distributed Heterogeneous Event Processing: Enhancing Scalability and Interoperability of CEP in an Industrial Context. *DEBS '10 Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems Pages 150-159*, 2010.

[31] N. P. Schultz-Møller, M. Migliavacca, and P. Pietzuch. Distributed Complex Event Processing with Query Rewriting. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 4:1–4:12, New York, NY, USA, 2009. ACM.

[32] Twitter. `http://twitter.com`.

[33] J. Urbani, A. Margara, C. Jacobs, S. Voulgaris, and H. Bal. AJIRA: a Lightweight Distributed Middleware for MapReduce and Stream Processing. *ICDCS*, 2014.

[34] J. Wolf, N. Bansal, K. Hildrum, S. Parekh, D. Rajan, R. Wagle, K.-L. Wu, and L. Fleischer. SODA: An Optimizing Scheduler for Large-Scale Stream-Based Distributed Computer Systems. *Middleware Lecture Notes in Computer Science Volume 5346, 2008, pp 306-325*, 2008.

[35] J. Xu, Z. Chen, J. Tang, and S. Su. T-Storm: Traffic-aware Online Scheduling in Storm. *ICDCS*, 2014.

[36] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized Streams: Fault Tolerant Streaming Computation at Scale. *SOSP*, 2013.

Reconsolidating Data Structures

Thomas Heinis[†] [*], Anastasia Ailamaki[‡]

[†] *Department of Computing, Imperial College, London, United Kingdom*

[‡] *Data-Intensive Applications and Systems Lab, École Polytechnique Fédérale de Lausanne, Switzerland*

ABSTRACT

For decades forgetting has been treated as an abnormality, a malfunction of the brain that leads humans to lose stored information. Recent results, however, suggest that forgetting is not only a malfunction of the human storage system, but also a useful feature. In order to guarantee a quick response in the face of the limited processing power of the brain, acting quickly on less or reduced information is key.

With storage becoming ever cheaper and continually growing it has become standard practice today to store each and every single data item. However, even increasingly powerful processors cannot deal with this data deluge. In this paper we consequently argue that forgetting and its mechanisms should be a part of today's data management, particularly for techniques requiring fast and/or approximate query answers. While forgetting or shedding information may have far-reaching implications for current methods in data management, in this paper we focus on discussing forgetting (and learning) in the context of data synopses.

1. INTRODUCTION

Experimental evidence in neuroscience research recently revealed that human forgetting [25] is not only a side effect of disease or of age, nor do we forget because the capacity of the brain is limited. Rather, we forget because the brain has limited computational power, yet we live in an environment where we need to make rapid judgments [1, 3]. The brain has consequently evolved to forget so that we can react quickly. In a world where we constantly absorb and learn information, we need to forget in order to enable split second decisions: we would rather react quickly based on imprecise or approximate information rather than too late. In an approximate world, who needs precision anyway?

One might argue, that forgetting is important in an environment with limited computational power and where approximate answers are sufficient, but that it has no place in a world with almost unlimited computational power. There are, however, many scenarios where computational power is limited or where response time is restricted. Forgetting can therefore also work to our advantage and in this paper we argue that forgetting or shedding information can also be beneficial in data management.

Forgetting may have broad implications and may trigger interesting discussions. Recent work, for example, introduces the notion of data rotting [18], i.e., a mechanism to periodically reduce data in a database to avoid it growing boundless (while ideally also adding the removed data in condensed and curated form back to the database). In this paper, however, we focus on the exact mechanisms of forgetting or, put in simple terms, fading memories (as well as learning) and use them to develop reconsolidating data structures, i.e., data structures that manage information through "forgetting" but also "learning". While we initially discuss the idea of reconsolidation structures broadly, we also elaborate on its application to data synopses in more detail. Surely, the case for data synopses [10] for quick and approximated answers has long been made, but what we discuss here is how to use some of the brains mechanisms to efficiently manage data synopses along with the resulting data management research challenges.

The remainder of this paper is structured as follows. We first discuss the neuroscience background of forgetting and learning in Section 2 and discuss where these mechanisms can fit into data management in Section 3. In Section 4 we elaborate on how the mechanisms of forgetting, i.e., reconsolidation, can be a powerful design principle when managing data synopses and discuss the resulting research challenges in Section 5. We discuss other potential applications in Section 6 and conclude in Section 7.

2. NEUROSCIENCE BACKGROUND

Several theories have been developed in recent decades to explain forgetting (as well as learning). For decades, the cognitive neuroscience theories of memory decay [4] (memory traces fade over time) and memory interference [16] (memory traces encoded with similar stimulus replace each other - similar to collisions in a hash map) were popular explanations for forgetting. More recent theories of consolidation [19] and reconsolidation [2], however, have gained much support lately, primarily due to their sound explanation of underlying cellular and molecular mechanisms [24]. They are today the most widely accepted explanations for forgetting (and learning).

In the following we discuss and summarise the consolidation/reconsolidation theories and discuss their implications.

[*]This work was done while the author was at EPFL.

10.5441/002/edbt.2015.66

2.1 Learning

The basic premise of learning is that memory traces [23] are not immediately stable or permanently stored when they are learned (encoded). Instead, memory traces are made permanent in the consolidation phase through strengthening (or weakening) the connections (synapses) between the neurons involved in a memory trace. By strengthening connections (or a synapses) between them, signals can be relayed quicker between two neurons and the firing patterns corresponding to memory traces can consequently be adapted.

The underlying biochemical process to strengthen synapses is very slow and can take several hours, during which memories are not stable. During this consolidation phase, the synapses need to be repeatedly stimulated. The hippocampus therefore repeatedly replays the firing patterns corresponding to the traces.

Any stress on the brain in general can interrupt the consolidation phase and memory traces may never become permanent in the brain or may only become weak. Anecdotal evidence for this interruption, for example, is that learned facts cannot easily be recalled after an all-nighter of learning (and implied lack of sleep). Similarly, interrupting consolidation with heavy drinking tends to erase whatever was recently experienced leading to a mental blackout.

2.2 Forgetting & Updating Memory

Although for a considerable time forgetting was assumed to be governed by completely different mechanisms, it is surprisingly similar to learning [2]. When retrieving memory traces (remembering), the traces become unstable or labile. The same synapses that were strengthened during the consolidation phase are rendered unstable and need to be reconsolidated in a process lasting for several hours similar to consolidation (despite built on different biochemical mechanisms).

Whilst retrieving information as a memory, the associated trace is therefore temporarily unstable and can be altered. Similar to interference during the consolidation process, also interference during reconsolidation alters the trace. Depending on the interference, whether it is positive or negative, memory can become stronger or weaker than it initially was. Reconsolidation has been extensively studied in fear memory, for example, in the experimental treatment of post-traumatic stress disorder, memories are recalled to make them unstable and interference in the form of electric stimulation is used to erase them. Similarly, positive feedback can be used to reinforce memory or to alter it [11]. Using positive feedback repeatedly can make memory traces more precise or can alter them substantially.

Experiments further show that the temporal dynamics of memory reconsolidation depend on the strength and age of the memory [20], such that younger and weaker memories are more easily reconsolidated than older and stronger memories. Similarly, the question whether old memory traces are updated or stored as new primarily depends on the age of the memory and also on the similarity of the event when recalling the trace: the similarity needs to be greater for updating an old memory, whereas for only recently encoded memory, the similarity threshold can be considerably smaller [9].

2.3 Reconsolidation in the Context of Data Management

To summarise from a computer science (or data management) perspective, forgetting is not simply a linear function of time. Instead, it is either a consequence of stress/disease or, surprisingly and crucially, from remembering it. Recalling memory makes the information unstable and requires reconsolidation. The reconsolidation process can lead to both, improvement (through updating) or to degradation (withering) if the reconsolidation process is interrupted. If and how much the memory is improved or degraded also depends on cognitive cues, i.e., feedback, during reconsolidation: positive feedback leads to improved memory whereas negative feedback generally leads to forgetting [11].

A key aspect of forgetting or learning memory is that traces are not completely erased (or completely accurately stored), i.e., they are not erased or learned as one, but are instead gradually degraded or improved. Thus, the mechanism of reconsolidation is a powerful means to reduce the amount of information that needs to be taken into account to answer a question (compute a query result) while still ensuring that relevant information is retained.

Today's ever-cheaper storage hardware allows storing nearly everything and eliminates the need of deleting old data. Similar to the brain, however, our ability to store data far exceeds today's data processing capacity and so, to keep answering queries quickly, we need to radically reduce the data taken into consideration. Using a mechanism similar to forgetting (and reconsolidation) in today's data management could be very powerful to gradually delete data permanently.

A more cautious approach to use the powerful mechanism of forgetting (and learning) in data management, however, does not permanently delete information/data. Instead it can be used to manage summaries of data by removing (forgetting) from summaries and adding data (from the full dataset) to them. The key aspect of forgetting (and learning) is how memory fades away (or is strengthened) gradually. This mechanism enables reducing the space needed to store a data structure by reducing its precision or expanding its size by increasing the precision. Consequently, its size and therewith query time cannot only be controlled by dropping or adding single items as a whole, but by reducing their size individually; similar to fading memories.

Using reconsolidation on proxy data structures has similarities to using caches in data management (and computer science in general) but also differs substantially.

First, caches are primarily small because they are expensive as opposed to the brain where the size is limited to guarantee response time. In the brain, forgetting is primarily driven by the need to reduce the time to process information. In the cache, however, the idea is rather to reduce communication time and the data is therefore moved to faster storage hardware (and also closer to processing). Essentially, it is not the processing time that is reduced, but rather the communication time, which helps in turn to reduce the overall processing time.

Second, in the case of reconsolidation, recently used data items are labile or prone to change whereas in caches, items that have not been accessed in a while are evicted and replaced by frequently and recently accessed data items (depending on the caching policy).

Third and crucially, items in a cache are typically either completely loaded or completely evicted (if room is needed for more items) and cache management is completely oblivious of the content of the items cached. Reconsolidation, on the other hand, is aware of the items' contents and reduces or improves their precision or resolution gradually.

With its ability to fade and strengthen data/information, reconsolidation is consequently particularly relevant for applications (or data structures) that allow for imprecision. This process is similar to data synopses [7], which can give quick approximate answers instead of accessing all data to give an excruciatingly slow (but precise) answer. Data synopses are pivotal today because the same technological advances that enable large-scale analysis of data also enable the generation of data on a similar scale. Yet the growing data generation capacity combined with cheap storage technology is likely to keep on outpacing the analysis capacity.

3. RECONSOLIDATING DATA STRUCTURES

What we contemplate here are reconsolidation data structures that, similar to caches or data synopses, act as a surrogate for the full dataset. Similar to the brain, they are limited by a time constraint to answer a query (and by giving a time constraint, they are also constrained by the space that they can use because of limited processing power available) and manage the precision of the data summary. The reconsolidation data structure, an approximated data summary, is used to quickly answer the query approximately.

At the core of reconsolidating data structure is the idea of treating data items as not oblivious to their content, but rather as aware of it. The major difference to existing caching strategies (and data synopses) is that data items are not dropped based on a binary decision, but their precision is decreased or increased, depending on how much the user is interested in them. Gradually improving or degrading the precision of data can of course only be tolerated in applications where precision is not crucial but where time matters. This is mostly true for applications where, similar to the brain, an approximate but quick answer is more important than an exact and slow one like, for example, in applications where humans consume the result.

To design reconsolidating data structures, we map the neuroscience mechanisms of forgetting and learning on the idea of data synopses and caches. This essentially means that we focus on a reconsolidating data structure (RDS) that acts as a surrogate for a complete (and potentially massive) dataset and is used to answer user queries. We further interpret queries to an RDS as a memory retrieval, i.e., recall of a memory trace. We then define accurate and complete queries to the underlying data structure dataset as new learning (or as interference with existing memory traces), i.e., improving precision of the information. Absence of a query to a dataset, on the other hand, is interpreted as if the answer from an RDS (and with it the RDS) is precise enough or, more precisely, too accurate and therefore uses more space than needed and its accuracy can thus be degraded. The age of memories in RDS is used to decide how fast the information is degraded: similar to the brain, old memories are degraded slower than new memories.

Put more simply: a query to the reconsolidation data structure answers the query and makes the data items touched labile/unstable. A subsequent query to the full data set means that the approximate query result was not precise enough and consequently learning starts, i.e., the precision of the data items touched is increased. Absence of a query to the data means that the approximate result was precise enough or, more importantly, too precise and the precision of the data items touched is reduced.

Of course application specific cues on the quality of the result can also be used to decide whether to improve or degrade precision, similar to Google's result ranking application where a click on a result is fed back to Google and is used to rank the results for the same keywords in future searches. Any such cue, however, is particular to an application and cannot be used in general.

Using a reconsolidation strategy for managing data synopses or caches, however, bears the risk that items are loaded into the surrogate dataset that are retrieved once but never again. Their precision may therefore never degrade and they may never be entirely dropped from the dataset. They will remain in the dataset indefinitely taking up space. However, we primarily propose reconsolidation data structures so that the response time in which approximate query answers are given is limited (and not to strictly adhere to a space budget). Still, if space is also a key concern, then additional mechanisms (like memory decay [4] where memory degrades as a function of time) can be used.

Clearly, for this idea to work in practice, the difficulty is to define the process of reducing or improving precision and to define how to query the resulting data structures. The latter, however, will in most cases remain the same. We will discuss this in the following for a number of applications where reconsolidation can prove useful.

4. EXAMPLE APPLICATION: DATA SYNOPSIS

The reconsolidation data structures we propose and discuss here are a rather abstract concept/mechanism that can be applied to different types of data structures. In the following we discuss how they can be used in the context of data synopses to make them a powerful tool in the face of a mass of ever growing data. We first provide background information on data synopses and then discuss the potential of reconsolidation for synopses.

4.1 Synopses Background

Research in data synopses has in the past primarily been driven by applications like data streams and cardinality estimation for query processing [7]. Their very nature, however, presents a great opportunity to accelerate approximate query execution in the context of big data.

4.1.1 Overview

The basic idea of data synopses is that the full data set is summarised, typically by using compression [7]. The synopsis acts as a surrogate of the data and is queried instead of the full dataset. Through compression or summarisation the synopsis is usually considerably smaller than the full dataset itself and, consequently, queries are executed substantially faster on the synopsis. The execution time of a query on the synopsis depends primarily on the size of the synopsis (unless additional auxiliary data structures like indexes are used). The size of the synopsis in turn depends on dataset characteristics, i.e., how easily compressible the data is, and is further controlled by the compression used.

Because of its substantial compression ratio, lossy compression is often used but doing so also leads to imprecise representations of the data. Data synopses based on lossy compression can consequently only approximately answer queries. Clearly there is a trade-off between size of the data synopsis (and thus query execution time) as well as the quality of the approximation, i.e., the smaller the approximation, the less accurate it is and thus the bigger the error becomes. Irregardless of the quality of approximation used, the key of data synopses is that they provide a user with tight error bounds expressing how accurate the received query result is.

Data synopses have in the past primarily been studied and used in the context of data streams and to estimate the cardinality of database tables (for query planning) [7]. With data growing beyond what can be today handled efficiently and reasonably, data synopses are again being considered as an interesting and competitive approach: instead of analysing the potential terabytes and petabytes of data in big data applications in a time consuming process, substantially smaller synopses can be queried almost instantly.

4.1.2 *Types of Synopses*

Considerable effort in the past has primarily developed four types of synopses. First, random sampling [21] takes samples at random out of the dataset (or the relation) and is very well suited for aggregate queries. Samples can be taken online, at query time from the full dataset, or offline, i.e., prior to querying to store them in a synopsis data structure. Online sampling is particularly interesting to improve the quality of the query result continuously: as long as the user is willing to wait, samples can be taken to improve the accuracy of the approximate query answer. For massive data that are primarily stored on disk, however, taking the samples online from the full dataset, as the query is being executed, is unlikely to be feasible due to the high cost of random access to the disk. Instead, big data sampling has to take samples offline, i.e., once from the full dataset and store the samples separately. Clearly, in the case of offline sampling, the more samples that are taken, the more precise the approximation, but the bigger the synopsis will also be.

A second well-researched type of synopsis is histograms [14]. In the context of databases, histograms play a crucial role in query optimisers and are often used for the purpose of data visualisation. Histograms summarise the data into bins each with its own value range, e.g., each bin stores the count of values/tuples in its range. Doing so makes them particularly useful for range-count queries, but they also have the potential to be used for general analysis queries [7].

Synopses based on wavelets summarise and approximate the data through wavelets [5]. Essentially, wavelet transformation is applied to relations or to time series resulting in a collection of wavelet coefficients. The size of the synopsis depends on how many coefficients are stored, which in turn defines the accuracy with which queries can be answered. The size of the synopsis alone, however, does not define the query execution time: at runtime, query execution can choose to ignore coefficients, thereby reducing query execution time, but also the degree of precision.

Relatively new are synopses based on sketches [6]. The basic rationale is to summarise the data per query type. As opposed to sampling, all data is considered, but only a small summary is retained (e.g., for a sum query all values are added up and only the sum is stored). As each query can be supported by a sketch, this approach is very powerful and applicable to all types of queries. Defining a new sketch per query type, however, requires considerable effort.

Artificial neural networks are also used to learn or approximate datasets [22] (for example time series [8]). Inspired by neuroscience, they use a graph with neurons as vertices and synapses as edges to answer queries approximately. Originating from machine learning neural networks are, however, rarely used as synopses in the management of data.

4.2 Reconsolidating Data Synopses

The basic idea of using reconsolidation data structures as a data synopsis [7] (or put more simply, using a synopsis featuring reconsolidation) is to improve or degrade the precision of the synopsis depending on the queries. The precision of regions frequently queried (in both, the synopsis and the dataset) is increased and the precision of those regions that are only queried in the synopsis is reduced.

A straightforward target to apply the idea of forgetting (or reconsolidation in general in order to support improving memory) is to use it on data synopses based on neural networks. They are modelled very similarly to the brain by using a graph with neurons as vertices and the connections (edges) between vertices represent the synapses. Synapse strength (and thus ultimately the encoding of information) can be modelled as weight of the edges. Degrading the synopsis is accomplished by reducing the weight of the edges (or by removing them altogether) and increasing the precision by increasing the weights. Changing the weight of the edges, however, will not effectively reduce the size of the synopsis. Hence, although neural networks lend themselves perfectly to the idea of forgetting, forgetting does not have a significant impact on the size of synopses based on neural networks or the time to execute queries.

There is, however, no need to restrict the idea of data synopsis or models based on neural networks. Much more interesting to apply reconsolidation to are data synopses based on, for example, wavelets [5]. Wavelets are used to approximately interpolate and therefore compress the underlying data set. Clearly the data synopsis is only an approximation of the real data set, but by using reconsolidation, areas or ranges of interest, i.e., queried over in the synopsis and in the complete dataset, can be stored with more precision. Others, retrieved only from the synopsis, can gradually be degraded by using less precise wavelets for interpolation.

Similarly, in data synopses based on histograms [10], precision can also be increased locally for interesting regions and can be decreased for uninteresting ones. In either case, the queries can be executed as usual on the reconsolidating synopsis. In any scenario where data synopses provide approximate answers, error bounds or guarantees are crucial.

5. DATA MANAGEMENT RESEARCH CHALLENGES

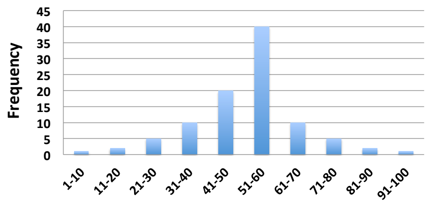
Applying the idea of reconsolidation to data synopses introduces several interesting data management challenges and thus research opportunities.

5.1 Adapting Data Synopsis Resolution

Key to the idea of reconsolidation data structures is to change the precision in given areas of it. Queries to the synopsis and the full datasets are used to infer what ranges (e.g., areas in a spatial model) the user is interested in exploring and analysing. The precision is increased in areas where the scientist is interested in and decreased elsewhere as a result of users' queries.

For sampling, assuming the samples are stored in a synopsis and are not taken online (in which case there is no data structure other than the full dataset needed), this can be achieved by taking and storing more samples from areas that users are interested in and deleting samples from areas where the interest is low.

In the case of histograms, improving precision is accomplished by adding bins and, thus, making the intervals (or value range of the bins) smaller and consequently more precise. Conversely, reducing the precision is similarly straightforward: neighbouring bins can be combined efficiently to make the resolution coarser (by increasing the value range) in areas with little interest. Figure 1 illustrates with the initial histogram (top) and the reconsolidated histogram (bottom) where in ranges of little interest (e.g., 1-30 & 70-100) bins are collapsed and in areas of a lot of interest bins are split.



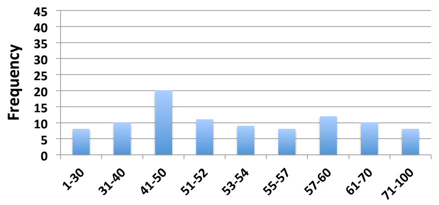


Figure 1: Initial histogram (top) and reconsolidated hisogram with adjusted precision (bottom).

For both, histograms as well as sampling, improving or reducing the precision is not very challenging. For wavelets and sketches, the two types of synopses that enable answers to more general classes of queries, however, changing the precision is not straightforward.

Wavelets are an interesting type of synopsis for reconsolidation. By definition, querying a wavelet based synopsis can be accelerated by ignoring coefficients that provide more precise query answers, thereby reducing the precision on the fly. Doing so, however, does not reduce the size of the synopsis itself and to apply the principle of reconsolidation, we can drop coefficients, in case there is little or no interest in an area, or learn and add coefficients (albeit in a computationally intense process from the full dataset) to the synopsis.

In the case of sketches, the precision is difficult to adjust. The challenge for sketches is that they are very application specific and it is thus difficult to find a generic way to define/implement their reconsolidation.

Adapting the resolution is consequently particularly challenging for wavelets and sketches where research has yet to develop efficient (for wavelets) and generic (for sketches) means to adapt to the precision. Research not only has to develop mechanisms to adapt the precision but also determine the methods to decide the exact area as well as the new level of precision.

5.2 Error Bounds for Variable Resolution

A key aspect of synopses is the idea to answer queries only approximately, but with tight error bounds. Providing the error bounds for query answers that only touch areas with the same precision is straightforward.

Given a synopsis with variable resolution, however, makes it challenging to compute the error bounds. Assume, for example, a synopsis based on histograms where the intervals in certain value ranges are smaller than in others, i.e., the interval length is variable. Clearly, ranges with smaller intervals have higher precision and thus smaller errors (and vice versa). The question, however, is how do we combine the different errors into a meaningful and intuitive error bound for a query that touches intervals of variable length?

A crucial research question, thus, is how to compute the error bounds based on a synopsis with variable precision, i.e., how to make the variable precision quantifiable or how to turn it into error bounds. Novel methods have to be developed to quantify the error bounds in case the ranges with varying precision are used.

5.3 Feedback Mechanisms

The feedback mechanisms described for data synopsis so far work by monitoring access to the full dataset. If the full dataset needs to be accessed, then the result (or the error bound) provided was not precise enough and consequently we have to learn, i.e., increase the resolution of the synopsis by taking more samples from the full dataset. If we do not have to access the full dataset, we can forget, i.e., the precision is decreased.

One research challenge consequently is whether better feedback mechanisms can be found. Clearly, for many applications better solutions can be used, e.g., using user input. Any such approach, however, is application specific and the research question is if generic mechanisms can be found to decide whether to learn or forget.

5.4 Answering Queries

The basic idea is for the user to gain as precise an answer as required from the data synopsis. If the answer, however, is not precise enough, then the full dataset is queried to provide a sufficiently precise answer (and also the data synopsis is improved through learning). The challenge to be addressed thus is how we can manage to only read the information additionally needed from the full dataset. This may be rather straightforward for wavelets, since in their case only additional coefficients can be read from disk (if they are stored on disk along with the full data) to make the result more precise.

For all other types of synopses the question of how to efficiently complement synopsis data with results, i.e., how to retrieve the minimal amount of information needed from disk and combine the results efficiently, is a challenging research question.

5.5 Data Organization

Changing the precision of the synopsis means either adding or removing data from it. A crucial research question directly affecting the performance of querying the synopsis is how to organise the data (presumably, given its size) on disk. Simply appending data when learning will lead to a data structure that requires excessive random disk access while only removing information (without reorganising the structure) means that considerable unnecessary data will be read and so the question becomes how can we design an updated efficient data structure for synopses?

6. OTHER APPLICATIONS

Reconsolidation has applications beyond data synopses and can be used in applications where imprecision can be tolerated or where data is imprecise/uncertain by nature.

6.1 Reconsolidating Caches

Clearly reconsolidation makes little sense if the data items in a cache are very small. In the case of hardware caches [13] (e.g., CPU caches) it is therefore unlikely to be used. Still for other types of caches in applications that can tolerate imprecision, reconsolidating can be used.

6.2 Content Distribution Networks

An interesting application for reconsolidation is caches in the context of content distribution networks [15, 17]. Objects in these caches are typically big and are consumed by users. A straightforward application is the degradation or improvement of image quality (or other multimedia content like videos or music) whether it is to return the image directly to the user or to query over it.

Degrading the image quality can be achieved by reducing the resolution, the size or by restricting the colour palette. Any of these approaches will reduce the effective size of the objects and thus query execution on the reconsolidation structure is accelerated. Imaging formats based on bitmaps or rasters may be difficult to degrade or improve precision easily and quickly, but there also exist layered formats where layers can be added or dropped individually.

7. CONCLUSIONS

What we present here is the need for forgetting and its mechanisms in the brain. We argue that forgetting should also have its place in data management. The brain, however, deals with imprecise information while many data management applications require precision and it is therefore not obvious where and how forgetting fits into data management. Given that storage is becoming ever cheaper, there seems to be little need to delete at all.

What we argue here, however, is that there is still a need to delete (or forget) to ensure timely query answers. This is particularly important as the quantity of data is growing quicker than the CPUs are becoming faster. Given the comparatively slow CPUs, we have to shed information to guarantee answers within a given time [12].

What we propose here is the mechanism or the design principle of reconsolidation that should be used in the design of applications and data structures. As we show with its application to data synopses, forgetting can be a powerful mechanism for managing data and yet entails considerable research challenges.

By discussing the example of data synopses we also demonstrate how the compelling mechanism of reconsolidation can be applied to applications where imprecision is acceptable.

Maybe the power of this approach is not so much in mapping the principle of reconsolidation strictly onto data management. However, we believe that reconsolidation (increasing or degrading precision) is a powerful mechanism, particularly for the management of data synopses and caches, which are becoming increasingly important to guarantee quick answers in face of today's (and tomorrow's) data deluge.

8. ACKNOWLEDGEMENTS

This work is supported by the Hasler Foundation (Smart World Project No. 11031) and the FET Flagship the Human Brain Project (grant agreement number 604102).

References

[1] J. Barrett and K. J. Zollman. The Role of Forgetting in the Evolution and Learning of Language. *Journal of Experimental Theoretical Artificial Intelligence*, 21(4):293–309, 2009.

[2] A. Besnard, J. Caboche, and S. Laroche. Reconsolidation of Memory: A Decade of Debate. *Progress in Neurobiology*, 99(1):61 – 80, 2012.

[3] R. A. Bjork and A. S. Benjamin. Successful Remembering and Successful Forgetting : a Festschrift in Honor of Robert A. Bjork, 2011.

[4] J. Brown. Some Tests of the Decay Theory of Immediate Memory. *Quarterly Journal of Experimental Psychology*, 10(1):12–21, 1958.

[5] K. Chakrabarti, M. N. Garofalakis, R. Rastogi, and K. Shim. Approximate query processing using wavelets. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00.

[6] G. Cormode and M. Garofalakis. Sketching Streams Through the Net: Distributed Approximate Query Tracking. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05.

[7] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. *Foundations and Trends in Databases*, 4(1):1–294.

[8] G. Dorffner. Neural Networks for Time Series Processing. *Neural Network World*, 6:447–468, 1996.

[9] S. J. Gershman, A. Radulescu, K. A. Norman, and Y. Niv. Statistical Computations Underlying the Dynamics of Memory Updating. *PLoS Computational Biology*, 2014.

[10] P. B. Gibbons and Y. Matias. New Sampling-based Summary Statistics for Improving Approximate Query Answers. In *Proceedings of SIGMOD '98.*

[11] M. J. Hays, N. Kornell, and R. A. Bjork. The Costs and Benefits of Providing Feedback During Learning. *Psychonomic Bulletin Review*, 17(6):797–801, 2010.

[12] T. Heinis. Data analysis: Approximation aids handling of Big Data. *Nature*, 515(7526):198, 2014.

[13] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., 2006.

[14] Y. Ioannidis. The History of Histograms (Abridged). In *Proceedings of the 29th International Conference on Very Large Data Bases*, VLDB '03.

[15] K. Johnson, J. Carr, M. Day, and M. Kaashoek. The Measured Performance of Content Distribution Networks. *Computer Communications*, 24(2), 2001.

[16] J. Jonides and D. Nee. Brain Mechanisms of Proactive Interference in Working Memory. *Neuroscience*, 139(1):181 – 193, 2006.

[17] J. Kangasharju, J. Roberts, and K. W. Ross. Object Replication Strategies in Content Distribution Networks. *Computer Communications*, 25(4), 2002.

[18] M. Kersten. Big data space fungus. In *Conference on Innovative Data Systems Research (CIDR '15).*

[19] J. L. McGaugh. Memory – a Century of Consolidation. *Science*, 287(5451):248–251, 2000.

[20] K. Nader. A Single Standard for Memory; the Case for Reconsolidation. *Debates in Neuroscience*, 1(1), 2007.

[21] G. Piatetsky-Shapiro and C. Connell. Accurate Estimation of the Number of Tuples Satisfying a Condition. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data.*

[22] D. F. Specht. A General Regression Neural Network. *IEEE Transactions on Neural Networks*, 2(6), 1991.

[23] L. Squire. Mechanisms of Memory. *Science*, 232(4758):1612–1619, 1986.

[24] N. C. Tronson and J. R. Taylor. Molecular Mechanisms of Memory Reconsolidation. *Nature Reviews Neuroscience*, (4):262 – 275, 2007.

[25] J. T. Wixted. The Psychology and Neuroscience of Forgetting. *Annual Review of Psychology*, 55(1):235–269, 2004.

A Generic Solution to Integrate SQL and Analytics for Big Data

Nick R. Katsipoulakis[1][*], Yuanyuan Tian[2], Fatma Özcan[2], Berthold Reinwald[2], Hamid Pirahesh[2]
[1]*University of Pittsburgh* katsip@cs.pitt.edu
[2]*IBM Almaden Research Center* {ytian, fozcan, reinwald, pirahesh}@us.ibm.com

ABSTRACT

There is a need to integrate SQL processing with more advanced machine learning (ML) analytics to drive actionable insights from large volumes of data. As a first step towards this integration, we study how to efficiently connect *big SQL* systems (either MPP databases or new-generation SQL-on-Hadoop systems) with distributed *big ML* systems. We identify two important challenges to address in the integrated data analytics pipeline: data transformation, how to efficiently transform SQL data into a form suitable for ML, and data transfer, how to efficiently handover SQL data to ML systems. For the data transformation problem, we propose an In-SQL approach to incorporate common data transformations for ML inside SQL systems through extended user-defined functions (UDFs), by exploiting the massive parallelism of the big SQL systems. We propose and study a general method for transferring data between big SQL and big ML systems in a parallel streaming fashion. Furthermore, we explore caching intermediate or final results of data transformation to improve the performance. Our techniques are generic: they apply to any big SQL system that supports UDFs and any big ML system that uses Hadoop InputFormats to ingest input data.

1. INTRODUCTION

Enterprises are employing various big data technologies to process huge volumes of data and drive actionable insights. Data warehouses integrate and consolidate enterprise data from many operational sources, and are the primary data source for many analytical applications, whether it is reporting or machine learning (ML). Traditionally data warehouses has been implemented using large-scale MPP SQL databases, such as IBM DB2, Oracle Exadata, TeraData, and Greenplum. Recently, we observe that enterprises are creating Hadoop warehouses in HDFS and Hadoop ecosystem, using SQL-on-Hadoop technologies like IBM Big SQL [13], Hive [21], and Impala [14]. In this paper, we use the term *big SQL systems* to refer to both the large-scale MPP databases as well as the SQL-on-Hadoop systems.

To gain actionable insights, enterprises need to run complex analytics on their warehouse data. There has been some works that embed ML inside SQL systems, through user defined functions (UDFs). We refer to this approach as the In-SQL analytics approach. Such examples include Hivemall [12] for Hive, and Bismarch [7] which is incorporated into the Madlib analytics library [11] for Greenplum and Impala [22]. However, through UDFs, only a limited number of ML algorithms can be supported. For example, only convex optimization problems can be implemented in Bismarch.

With the big data revolution, most new developments of big ML algorithms happen outside the SQL systems, and mainly on big data platforms like Hadoop. There are many options, such as MLLib [20], SystemML [9], and Mahout [1], and more systems and special algorithms are developed every day. Enterprises need to integrate their big SQL system with their big ML system. The solution should also be extensible to any future system. The following example will demonstrate this need of integration.

An Example Scenario. A data analyst from an online retailer wants to build a classification model on the abandonment of online shopping carts in USA. The detailed information of online shopping carts and customers are stored in two tables `carts` and `users` either in a MPP database or in a SQL-on-Hadoop system. To prepare the data that she later will feed into an SVM algorithm, the analyst needs to combine the two tables and extract the three needed features, the customer's `age`, `gender` and the dollar `amount` of the shopping cart, as well as the indicator field `abandoned` for building the classification model. This data preparation can be easily expressed as a SQL query shown below.

```
SELECT U.age, U.gender, C.amount, C.abandoned
FROM carts C, users U
WHERE C.userid=U.userid AND U.country='USA'
```

Spark provides a unified environment that allows combining SQL (Spark SQL) and ML (MLlib) together. The data handover between Spark SQL and MLlib is through the distributed (and often in-memory) data structure, called Resilient Distributed Datasets (RDDs). Again, analysts are limited by the ML algorithms supported in MLlib. If an analyst wants to use an existing algorithm in Mahout or if she has her own analytics algorithm already implemented in MapReduce, she has to write the data into HDFS, run her analytics algorithm, and store results back into HDFS. In addition, in both Spark and the In-SQL analytics approach, one is locked in a particular environment. But in reality, enterprises need a generic solution that works with many big SQL and big ML system, and is easily extensible to any future system.

The straightforward approach to connect big SQL and ML system is through files on a shared file system, such as HDFS since most big ML systems are running on Hadoop. In other words, the big SQL system outputs results onto HDFS and then the big ML system reads them from HDFS. This approach obviously incurs a lot of overhead. In this paper, we explore whether we can do better

[*]The work described in this paper was done while the author was working at IBM Almaden Research Center

than this basic approach.

We identify two major challenges when connecting big SQL and big ML systems: (1) data transformation and (2) data transfer.

Data transformation deals with the fact that SQL systems and ML systems prefer data in different formats. For example, most ML systems work on numeric values only. For categorical values (e.g. the gender of a customer) normally stored as strings in SQL systems, they have to be *recoded* [5] and sometimes *dummy coded* [4] (details will be provided in Section 2) before the analysis can be applied. Today, such transformation functionalities are rarely provided in either big SQL or big ML systems, which means they have to be implemented by users. One can choose to implement these transformation functions outside both systems, using her preferred data transformation framework, e.g, MapReduce. But, this introduces another hop, hence extra overhead, between SQL and ML systems. A better approach would be to incorporate transformations in either SQL or ML systems. In order to provide a generic solution, we leverage the extensibility (e.g UDFs) of big SQL systems and propose an *In-SQL transformation* approach. In fact, we found that most common data transformations between SQL and analytics can be implemented through UDFs by exploiting the massive parallelism inside big SQL systems.

Data transfer, on the other hand, deals with how the output of a SQL system is passed over to the ML system for processing. When the SQL system requires a long haul to produce the output, the straightforward approach of handing over files on a shared file system may be preferred for fault tolerance reasons. But the extra file system write and read can be a performance hurdle. Another important issue about the straightforward approach is the fact that the entire output of the transformation step needs to be produced and materialized (a blocking operation) before it can be ingested into the ML system. In this paper, we propose a general approach to *parallel data streaming* between these two systems. This approach avoids touching the file system between SQL and ML systems, and can be used by any big SQL system that supports UDFs and any big ML system that uses Hadoop InputFormat for ingesting data (in fact, all ML systems on Hadoop do) in parallel.

Caching is a common technique used in distributed systems to reduce communication costs. In this paper, we explore caching intermediate or final results of the transformation step to help significantly reduce the costs of connecting big SQL systems with big ML systems. Most often the intermediate results that are required by the recoding transformations can be precomputed and reused.

Note that the need for integrating SQL and ML existed even before the era of big data. People have been fetching data from databases and feeding them to ML softwares, such as R [19], in the past. But since the data exchanged between the two systems were small, data transformation and transfer were not challenging problems. For example, there are functions provided in R (sequential implementations) for common data transformations. Data transfer, on the other hand, is usually done through passing physical files around. However, this old way of sequentially transforming data and passing files around is often infeasible when huge volumes of data are involved. Exploiting the massive parallelism inside and between big SQL and big ML systems is a necessity to guarantee performance, and this is exactly what we strive to achieve in this paper.

The contributions of this paper are as follows:

- We first propose an In-SQL approach to incorporate common data transformations for analytics inside big SQL systems through UDFs, by fully exploiting the massive parallelism of the big SQL systems.
- We then introduce a general approach to transfer data between big SQL and big ML systems in a parallel streaming fashion, without touching the file system.
- We further explore caching techniques to reduce the costs of connecting big SQL and big ML systems.

2. IN-SQL DATA TRANSFORMATION

Most big SQL systems today have UDF support for extensibility, which makes it feasible to employ a generic In-SQL solution for common transformations for ML. We will use the two most common transformations, recoding of categorical variables and dummy coding, as examples to demonstrate how these transformations can be implemented in parallel fashions using UDFs. Some less common transformations, such as effect coding and orthogonal coding [6], can be implemented in similar ways as dummy coding.

2.1 Recoding of Categorical Variables

Most data transformation between SQL and ML systems deals with categorical variables. This is because categorical variables are usually represented as string fields in SQL systems, but it is very hard and inefficient to handle string values in analytics. As a result, most ML systems prefer handling numeric values only. One of the most common data transformation, therefore, is recoding of categorical variables [5]. Figure 1(b) shows an example recoding of the categorical fields `gender` and `abandoned` in the table shown in Figure 1(a) (This table could be the result of a query in a SQL system). The recoded numeric values are usually consecutive integers starting from 1. Here, for the field `gender`, the value ‘F’ is recoded to 1 and ‘M’ is recoded to 2. And for the field `abandoned`, ‘Yes’ and ‘No’ are recoded to 1 and 2 respectively.

The above recoding is seemingly simple. In a centralized environment, it only requires one pass of data to perform the recoding of all categorical fields, assuming the number of distinct values for each field is not large. This centralized algorithm simply keeps track of a running map of current recoded values for each categorical field while scanning through the data. If a value of a field has been seen before, it just uses the map to recode it, otherwise a new recoding is added to the map.

In a distributed environment, however, a two-phase approach is needed. In the first phase, each local worker computes its distinct values for each categorical field in its local partition, and then exchanges the local lists to obtain the global distinct values. In the second pass of the data, we can use the global distinct values to perform the recoding.

This two pass algorithms can be easily implemented using a combination of UDFs and SQL statements. For example, in the first pass, we can implement a parallel table UDF, which in parallel reads its local partition of the table and generate another table with two fields `colName` and `colVal`, which contains the local unique values for each categorical column. For example, the returned records in a local partition might be `{(‘gender’, ‘F’), (‘gender’, ‘M’), (‘abandoned’, ‘Yes’)}`. These records can then be passed to a `SELECT DISTINCT colName, colValue FROM ...` statement to compute the global unique values. We can also introduce another table UDF to add a recoded value field `recodeVal` to the results, generating recode mapping records like `{(‘gender’, ‘F’, 1), (‘gender’, ‘M’, 2), (‘abandoned’, ‘Yes’, 1), (‘abandoned’, ‘No’, 2)}`. Let’s denote the original table as `T` and the recode map table as `M`, then the final recoding in the second pass can be simply implemented by a join like below:

age	gender	amount	abandoned
57	'F'	108.00	'Yes'
40	'M'	57.98	'Yes'
35	'F'	265.97	'No'

(a) original table

age	gender	amount	abandoned
57	1	108.00	1
40	2	57.98	1
35	1	265.97	2

(b) recoding

age	female	male	amount	abandoned
57	1	0	108.00	1
40	0	1	57.98	1
35	1	0	265.97	2

(c) dummy coding

Figure 1: Recoding and dummy coding of categorical variables

```
  SELECT T.age, Mg.recodeVal as gender, T.mount,
Ma.recodeVal as abandoned
FROM T, M as Mg, M as Ma
WHERE Mg.colName=‘gender’ AND T.gender=Mg.colVal
AND Ma.colName=‘abandoned’ AND T.abandoned=Ma.colVal
```

Although one could use SQL queries to compute the distinct values, each column that needs to be recoded would result in such an SQL query, and would require one pass of the data. Using UDFs, we can scan the data once and compute the distinct values for all required columns.

Note that although categorical values are represented as strings in tables, some modern column stores are able to exploit dictionary compression to physically store string values as integers. Utilizing these integers directly as the recoded values for ML systems is an interesting direction. However, there are a number of challenges. First of all, the internal physical dictionary encoding is usually not exposed to users, thus utilizing the encoded integers is difficult or even impossible in a general approach using UDFs. Second, most dictionary compression for big SQL systems, such as in the Parquet format [18] for Impala and ORC format [17] for Hive, is applied only for a local partition of data. Therefore, we cannot directly use the local encoded integers for the global recoding. Lastly, some ML systems, such as SystemML [9], require the recoded categorical values to be consecutive integers starting from 1. Some dictinary compression algorithms may not produce consecutive integers. Moreover, the recoding needs to be done on filtered data, and hence we may have to recode the values again.

2.2 Dummy Coding

Some ML algorithms, such as SVM and logistic regression, require generating binary features from a categorical variable before invoking the algorithms. This transformation is called dummy coding [4]. People also call it one-hot encoding or one-of-K encoding. Figure 1(c) shows an example dummy coding for the `gender` field in the recoded table of Figure 1(b). In dummy coding, a categorical variable with K distinct values is split into K binary variables. Assuming the categorical variable has already been recoded, then the original variable with value i results in the i^{th} binary variable to be 1, and the remaining $K-1$ variables to be 0.

To implement dummy coding in big SQL systems, we only need a parallel table UDF that takes in the number of distinct values for each categorical variable (already obtained during recoding phase) and scans through each partition to perform the dummy coding in parallel.

3. PARALLEL STREAMING DATA TRANSFER

In this section, we describe our approach to parallel streaming data transfer. There are two main goals that we want to achieve when designing the streaming data transfer method: (1) generality of the approach on various big SQL and big ML systems, and (2) exploitation of the massive parallelism between the two systems.

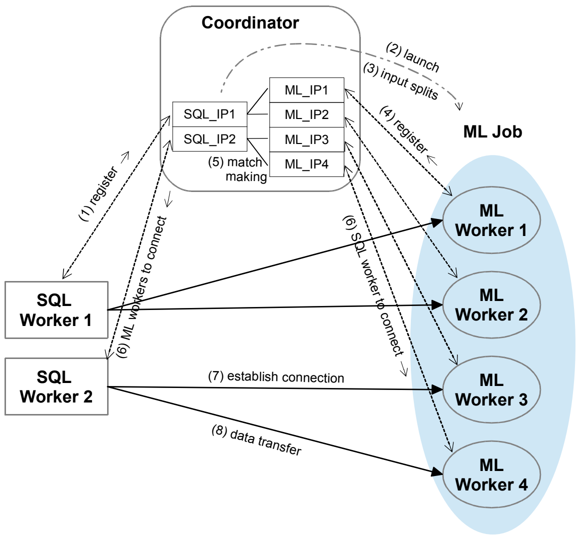


Figure 2: Information and data flow in parallel streaming data transfer

To achieve generality, we again exploit the UDF extensibility in the big SQL systems and extend the Hadoop's InputFormat interface in the big ML systems. In fact, most existing big ML systems [9, 1, 20] can input data through the InputFormat interface. So, a user can choose any of the existing big ML system to run the analytics. The only change she has to make is to use our specialized SQLStreamInputFormat in the job configuration. To exploit the massive parallelism between two independent distributed systems, we introduce a long standing *coordinator* service to help bridge the two systems to establish parallel communication channels. In addition, we try to take advantage of data locality as much as possible when the big SQL and big ML systems share the same cluster resources.

Figure 2 shows the detailed information and data flow in our parallel data streaming method. The data transfer starts from the parallel table UDF in the SQL system. This UDF takes in as inputs the table to be transferred, the IP and port number of the coordinator, as well as the command and arguments to invoke the desired algorithm of the target ML system. When this UDF is executed in each SQL worker, it first connects to the coordinator, notifies the coordinator of its own worker id, IP address, and the total number of active SQL workers, and also passes along the command and arguments of the target ML algorithm (step 1 in Figure 2). When all the SQL workers have registered, the coordinator launches the ML job with the provided command and arguments (step 2).

When the ML job tries to spawn tasks to read data, it first creates an InputFormat object. InputFormat has a member function called getInputSplits(), which is responsible for dividing the input data into subsets. Each subset is called an InputSplit and is consumed by one ML worker. In other words, the number of InputSplits equals to the number of ML workers. We customized the getInputSplits method to contact the coordinator to decide on the InputSplits (step 3). Let n be the number of SQL workers and m be the number of InputSplits. If m is not pre-specified by the particular ML algorithm, then we always set $m = n \times k$, where k is a parameter to control the degree of parallelism in the ML job. We divide the needed m InputSplits evenly into n groups, with each group corresponding to the data from one SQL worker, as demonstrated in Figure 2. To take advantage of the potential locality, we also provide the locations for each InputSplit where the data for the split would be local. In particular, for each InputSplit corresponding to the i^{th} SQL worker, we use the IP address of this SQL worker as the location of the InputSplit. With the provided locations, when the ML job spawns the ML workers to read data, it tries to colocate, when possible, in a best effort manner, the ML workers with the corresponding SQL workers, so that data transfer does not incur network I/O.

After the ML workers are spawned, they register themselves back to the coordinator (step 4 in Figure 2). Then, the coordinator matches the IP of each SQL worker with the IPs of its corresponding ML workers (step 5), and subsequently sends the matched information back to the workers on both sides (step 6). Now, the job of the coordinator is done. Finally, the SQL workers and the ML workers establish the TCP socket connections (step 7), before the actual data transfer starts (step 8). Each SQL worker sends data to its ML workers in a round robin fashion. Inside a SQL worker, there is a send-buffer associated with each target ML worker for buffering the sent data. Similarly, each ML worker has a receive-buffer to buffer the received data from its corresponding SQL worker. The sizes of the buffers are controllable system parameters. If an ML worker is slow to ingest its data and the corresponding send buffer becomes full, we can spill it onto the local disks to syncronize the producer and consumers.

4. QUERY REWRITER FOR DATA TRANSFORMATION AND TRANSFER

Although we have provided various UDFs for the common data transformation and parallel data streaming, it is still a difficult task for the users to compose the queries to invoke these UDFs. For ease of use, we provide a query rewriter outside the SQL systems. A user provides this query rewriter with her SQL query (such as the example query in Section 1), the transformations needed on the results of the query, and if parallel data streaming is needed, the necessary information for calling the target ML algorithm. Then, the query rewriter will extend the given query into another query with UDFs, and other operations to perform the required transformations and the data transfer.

5. CACHING

When similar data transformations are repeated between a big SQL system and a big ML system, we can exploit caching to reduce the cost. We identified two cases of caching, assuming there is no data update: (1) caching fully transformed data, and (2) caching intermediate recode maps.

5.1 Caching Fully Transformed Data

In this case, we cache the fully transformed data in the big SQL system by storing it as a materialized view or an actual HDFS table. If later another ML algorithm needs to be run on the data resulted from the same SQL queries, we can directly reuse the stored data, thus saving the cost of the SQL queries and the data transformation all together. This situation happens, for example, when an analyst wants to run a number of classification algorithms, such as SVM, logistic regression, naive Bayes and decision trees, to compare the quality of different classifiers on a particular dataset.

Besides the above case, the fully transformed data can also be reused if a subset of the transformed data is needed. Let's take the example scenario in Section 1 as an instance. If we cache the fully transformed result of this query, and later we encounter another query shown below as the data preparation for an ML algorithm, we can fully utilize the cached data, without running the query and transforming the query result.

```
SELECT U.age, C.amount, C.abandoned
FROM carts C, users U
WHERE C.userid=U.userid AND U.country='USA'
AND U.gender = 'F'
```

The reason that we can fully utilize the cached result is that this new query satisfy the following conditions:

1. It contains the same tables in the from clause, and the same join conditions and predicates in the where clause, as the query for the cached data.
2. The projected fields are a subset of the projected fields in the query for the cached data.
3. Additional conjunctive predicates are only on the projected fields in the query for the cached data.

In fact, if we denote the result of query in Section 1 as `T`, the new query can be expressed as a selection and projection query on table `T` as below.

```
SELECT age, amount, abandoned
FROM T
WHERE gender = 'F'
```

5.2 Caching Recode Maps

The applicability of caching the fully transformed data is limited. For the following query, the cached data cannot be used at all, as it does not satisfy the conditions described in the previous subsection.

```
SELECT U.age, U.gender, C.amount, C.nItems, C.abandoned
FROM carts C, users U
WHERE C.userid=U.userid AND U.country='USA'
AND C.year = 2014
```

However, we notice that this query satisfies a different set of conditions, which allow it to benefit from caching the intermediate recode map (see Section 2.1) generated during the transformation of the previous query:

1. It contains the same tables in the from clause, and the same join conditions in the where clause, as the previous query.
2. It contains predicates on the same set of fields as the predicates on the previous queries, and each predicate is either the same as or logically stronger than (e.g. $a < 18$ is logically stronger than $a \leq 20$) the corresponding predicate in the previous query.
3. The projected categorical fields are a subset of the projected categorical fields in the previous query.
4. Additional predicates are conjunctive.

By reusing the recode map for the new query, we avoid one of the two passes for the new query during recoding.

As can be seen above, the way we detect whether a query can benefit from the cached data is similar to utilizing materialized views in query optimization [10, 16] and we extend the query rewriter introduced in Section 4 to utilize these techniques. When the rewriter gets a query, it first check to see if any of the existing materialized views can be used and rewrites the query accordingly.

6. DISCUSSION ON FAULT TOLERANCE

Fault tolerance is a very hard problem for integrating big SQL and big ML systems. First of all, if either the underlying big SQL system or the big ML system lacks fault tolerance support, the whole integration pipeline has to be restarted from scratch in case of a failure. In fact, most MPP databases do not support mid-query failure recovery. Most SQL-on-Hadoop engines, like Impala [14] also sacrifice mid-query recory. As for big ML systems, MLlib is the only one known to support mid-query fault tolerance. Even both underlying systems provide fault tolerance guarantees, we still need the connection between the two to be resilient to failures. If data transfer between the two systems is through files on HDFS, or if the cached results from the big SQL system can be directly reused by the big ML system, the fault tolerance can be guaranteed. On the other hand, if parallel data streaming in Section 3 is used, more care has to be taken. First, we need the coordinator service to be resilient itself. This can be achieved by using Zookeeper [2]. In addition, when the data transfer between a SQL worker and an ML worker fails, due to the failure of either end points or the connection, we need to notify the big SQL system to restart the SQL worker and simultaneously tell the big ML system to restart all the ML workers corresponding to the SQL worker, so that the data transfer can be resumed.

An alternative to our streaming data transfer is utilizing an in-memory file system like Tachyon [15], which would provide fault-tolerance guarantees. However, Tachyon is still very Spark and RDD oriented, whereas in this work we thrive to be provide a generic solution that works for all big SQL and big ML systems, whether they run their own native processes, or use MapReduce or Spark.

7. PRELIMINARY EXPERIMENTS

In this section, we report the results of our preliminary experimental study by using IBM Big SQL 3.0 as the big SQL system and Spark MLlib as the big ML system.

We used 5 servers for all our experiments. Each had 2x Intel Xeon CPUs @ 2.20GHz, with 6x physical cores each (12 physical cores in total), 12x SATA disks, 1x 10 Gbit Ethernet card, and a total of 96GB RAM. Each node runs 64-bit Ubuntu Linux 12.04, with a Linux Kernel version 3.2.0-23. One of the servers was used as the HDFS NameNode, MapReduce JobTracker, Spark master as well as Big SQL head node. The remaining 4 servers host the HDFS DataNodes, MapReduce TaskTrackers (9 mappers per server), Spark workers (6 workers on each server) and Big SQL workers (1 worker with multi-threading on each server). HDFS replication was set to 3. The send-buffer and receive-buffer sizes were both set to 4KB for the parallel data streaming.

We generated synthetic datasets in the context of the example query scenario described in Section 1. In particular, we created a 56GB carts table with 1 billion records and 361 MB users table with 10 million records. Both tables were stored in text format on HDFS. We ran the SQL query shown in the example in Section 1, transformed the result (recoding the categorical variables and dummy coding), and passed the result to MLlib for running the SVMWithSGD algorithm. In our experiments, we report the time for processing the SQL query, transforming the result, transferring the transformed data to the ML job, and reading the input data in

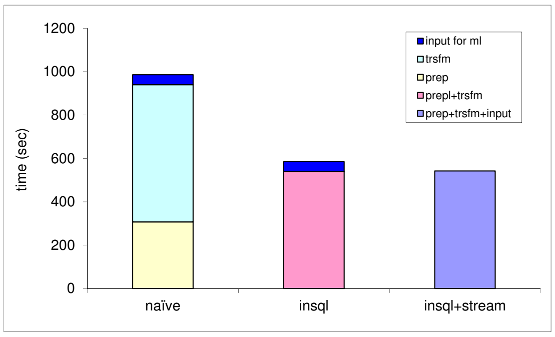


Figure 3: Comparison of three approaches of connecting big SQL and big ML systems

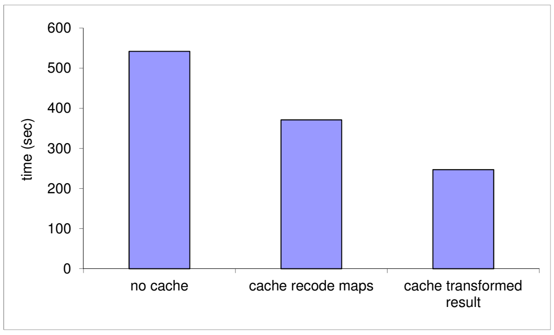


Figure 4: Effect of caching

the ML job. We do not report the runtime of the ML algorithm, because it is highly dependent on the actual data and the algorithm (e.g how many iterations to converge). For example, reading the transformed data from HDFS and running the SVMWithSGD for 10 iterations took 774 seconds. In the ML job, we first read the input data, whether it is from HDFS or from parallel data transfer, into a Spark in-memory RDD. After that we pass the RDD to the MLlib SVMWithSGD algorithm. The measured time of reading input in the ML job is the time from the start of the job till the in-memory RDD is constructed.

Figure 3 compares three approaches of connecting SQL with ML for big data. In the naive (denoted as naive in Figure 3) approach, we use Big SQL to execute the SQL query and materialize the result on HDFS (prep). Then, we use a third tool, Jaql [3], to perform the data transformation, since Jaql has built-in functions for recoding of categorical variables and dummy coding (trsfm). Again, the result is written to HDFS. Finally, Spark MLlib reads the data form HDFS (input for ml) and performs the ML job. The breakdown of execution times for different stages is shown in the figure. The second approach, denoted as insql in the figure, employs the In-SQL transformation method (we implemented the recoding of categorical variables and dummy coding in Big SQL using UDFs). In this approach, the transformation is combined together with the SQL query, thus the operations can be performed in a pipeline (prep+trsfm). In the third approach, denoted as insql+stream, we use the parallel streaming data transfer in addition to the In-SQL transformation. Now, all operations can be pipelined together (prep +trsfm+input). In addition, as we use in-memory RDDs to store the input data in Spark for the ML job, the transformed data from Big SQL is never written to HDFS. As can be seen from Figure 3, the In-SQL transformation significantly improves the performance of the whole work flow: 1.7x speed up against the naive approach. The insql+stream approach further improves the performance by another 43 seconds. This is a significant reduction in data ingestion cost in Spark MLlib (reading from HDFS takes 46 seconds), although it is not an impressive number in the overall workflow. In this particular case, the transformed data itself was not very large (5.6GB), and hence reading it from HDFS was not dominating the overall data pipeline. If a larger dataset were used, the performance would be more dramatic. We also want to note that if the ML algorithm takes a long time to produce the desired model, then whether using HDFS or streaming for data transfer makes little difference in the overall performance. In addition, HDFS can also provide additional fault tolerance, and could be preferable if the ML system requires reading the input multiple times.

We now investigate the effect of caching intermediate or final results of data transformation in connecting big SQL and big ML systems. We compare the In-SQL transformation approach with the approach where we cache intermediate recode maps, and the approach where we cache the fully transformed result in Figure 4. In all three approaches, we employ the parallel streaming transfer to pass data to the ML job. Evidently, if caching can be used, then the fully cached result will provide the best performance (2.2x speedup against no cache), followed by the cached intermediate recode maps (1.5x speedup against no cache). But, keep in mind that the applicabilities of the two caching approaches are limited, with caching the fully transformed result most limited. For both caching methods, the reusability depends on the complexity of the preparation SQL query and how fast, or if at all, the data gets updated.

8. CONCLUSION

In this paper, we studied the problem of integrating SQL and ML processing for big data, providing a general purpose solution that will work with *any* big SQL system that supports UDFs and *any* big ML system that uses InputFormats to ingest its input. We focused on two problems: data transformation and data transfer between the two systems. In particular, we proposed an In-SQL approach to incorporate common data transformations for ML algorithms inside big SQL systems through UDFs. In addition to the basic approach of using files as the media for data transfer between systems, we proposed a general streaming data transfer approach by introducing UDFs in big SQL systems and implementing a special Hadoop InputFormat. Furthermore, we explored the use of caching intermediate or final results of the transformations to reduce the costs of connecting SQL and ML systems. Our preliminary experimental results show that the In-SQL approach has great potential in reducing the data transformation cost, and caching is very effective in improving the performance of the whole analytics work flow. The parallel streaming data transfer approach has its pros and cons, depending on the target ML system. Our preliminary experiments show it results in significant reduction in data ingestion costs for MlLib, which uses in-memory RDDs.

As future work, we plan to investigate using a message passing system like Kafka [8] to pass the data between SQL and ML workers. Kafka would gurantee at least one read, in case of failures. Kafka could also be the system to cache the data when the ML workers are not fast enough to consume the data. We also plan to build a generic data exchange infrastructure that utilizes memory and streaming between different frameworks running on big data platforms, such as streaming, batch, SQL, ML, etc.

9. REFERENCES

[1] Apache Mahout. `https://mahout.apache.org`.

[2] Apache ZooKeeper. `http://zookeeper.apache.org`.

[3] K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Y. Eltabakh, C.-C. Kanne, F. Özcan, and E. J. Shekita. Jaql: A scripting language for large scale semistructured data analysis. *PVLDB*, 4(12):1272–1283, 2011.

[4] D. Conway and J. M. White. *Machine Learning for Hackers*. O'Reilly Media, 2 2012.

[5] Cookbook for R: Recoding data. `http://www.cookbook-r.com/Manipulating_data/Recoding_data`.

[6] Dummy, effect, & orthogonal coding. `http://luna.cas.usf.edu/~mbrannic/files/regression/anova1.html`.

[7] X. Feng, A. Kumar, B. Recht, and C. Ré. Towards a unified architecture for in-rdbms analytics. In *SIGMOD*, pages 325–336, 2012.

[8] R. C. Fernandez, P. Pietzuch, J. Koshy, J. Kreps, D. Lin, N. Narkhede, J. Rao, C. Riccomini, and G. Wang. Liquid: Unifying Nearline and Offline Big Data Integration. In *CIDR*, 2015.

[9] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. SystemML: Declarative machine learning on mapreduce. In *ICDE*, pages 231–242, 2011.

[10] A. Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, 2001.

[11] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The madlib analytics library: Or mad skills, the sql. *PVLDB*, 5(12):1700–1711, 2012.

[12] Hivemall. `https://github.com/myui/hivemall`.

[13] IBM Big SQL 3.0: Sql-on-hadoop without compromise. `http://public.dhe.ibm.com/common/ssi/ecm/en/sww14019usen/SWW14019USEN.PDF`.

[14] Impala. `http://github.com/cloudera/impala`.

[15] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *SOCC*, 2014.

[16] I. Mami and Z. Bellahsene. A survey of view selection methods. *SIGMOD Rec.*, 41(1):20–29, 2012.

[17] The ORC format. `http://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.0.0.2/ds_Hive/orcfile.html`.

[18] The Parquet format. `http://parquet.io`.

[19] R. `http://http://www.r-project.org/`.

[20] Spark MLlib. `https://spark.apache.org/mllib`.

[21] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[22] How-to: Use MADlib pre-built analytic functions with impala. `http://blog.cloudera.com/blog/2013/10/how-to-use-madlib-pre-built-analytic-functions-with-impala`.

ECCO- A Framework for Ecological Data Collection and Management Involving Human Workers

Senjuti Basu Roy[†], Sihem Amer-Yahia[◇], Lucas Joppa[††].
[†]UW Tacoma, [◇] CNRS, LIG, [††] Microsoft Research
senjutib@uw.edu,sihem.amer-yahia@imag.fr,lujoppa@microsoft.com

ABSTRACT

Scientific and ecological data collection in today's world is primarily driven by citizen-based observation networks to gather information on a diverse array of species and natural processes. Such efforts leverage the contributions of a broad recruitment of human observers to collect data and use Machine Learning algorithms to process the collected data leading to a computational power that far exceeds the sum of the individual parts. Instead of organic group formation and collaboration, our vision is the need to formalize collaboration and rethink the components of a data management system to ensure its sustainability in such human-intensive applications. The enabler of collaboration is the notion of *a user group* that implies different behaviors and interactions between its members. We advocate the design of new components of a data management system that deliberately acknowledge the uncertainty and dynamicity of *human behavior* by capturing the *human factors* that characterize group members. We describe ECCO, a framework that contains two generic components: *adaptive collaborative human factors learning* and *adaptive human-centric optimization*. Those are the core components that support the fundamental functionalities of a wide range of human-intensive applications. ECCO components rely on two optimization engines, namely *task assignment* and *human data management engine*. An additional challenge in designing the components of ECCO is the need to support adaptive and incremental computation. We discuss the modeling, learning, and computational challenges of designing the components of ECCO and propose a roadmap of future directions of this vision.

1. INTRODUCTION

Achieving insight about ecological patterns often requires the study of natural systems at large scales. An emerging focus therefore is to build an infrastructure for data synthesis and analysis that allows data collection and organization across the continent and perform large scale analyses over it. While new technologies are gradually emerging to

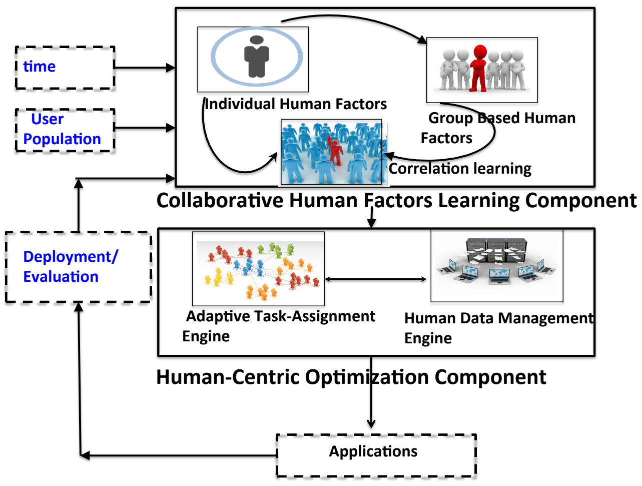

time
User Population
Individual Human Factors
Group Based Human Factors
Correlation learning
Collaborative Human Factors Learning Component
Deployment/ Evaluation
Adaptive Task-Assignment Engine
Human Data Management Engine
Human-Centric Optimization Component
Applications

Figure 1: High Level Design of ECCO

leverage autonomous sensor networks for such data collection, the state of the art techniques still can not identify organisms to species, they serve to gather information on the variables that influence species occurrence. Therefore, most data on species-level occurrence still must be gathered by humans [10], necessitating innovative programs for wide-scale data collection and analysis. In particular, the ultimate objective of such effort is to build a hybrid human machine computational power to solve complex problems. We advocate the design of a new framework ECCO to that end with the vision to formalize collaboration and rethink the components of a data management system to ensure quality and sustainability in such human-intensive applications.

Applications: Several leading efforts of citizen science are being carried out nationally and internationally. For example, the US National Phenology Network[1] conducts Project Budburst, a citizen-based effort to report phenological events such as first leafing, first flowering, and first fruit ripening for a variety of plant species in order to better understand the broad scale effects of climate change. The Galaxy Zoo [2] project provides access to almost 250,000 images of galaxies and engages volunteers to classify them into shapes in order to better understand how galaxies are formed. In FoldIt [3] project, researchers attempt to predict

[1]http://www.usanpn.org/
[2]http://www.galaxyzoo.org/
[3]http://fold.it/portal/

the structure of a protein by taking advantage of puzzle solving abilities of the human. Another popular example is the e-Bird project [11, 4], which engages a vast network of human observers (citizen-scientists) to report bird observations using standardized protocols.

Objective: The ultimate objective of such efforts is to employ a hybrid human and mechanical computation power to solve complex problems through active learning and feedback. The contributed large scale data is processed with Machine Learning algorithms for correlating species distributions with environmental covariates to identify the unlabeled points that when labeled would most rapidly decrease uncertainty in the model being deployed, or contain the highest amount of surprise versus expectation, or most likely result in a different model being proposed. To seamlessly enable such capabilities to the domain scientists, therefore, the challenge is to develop the appropriate data management and optimization framework that will allow effective group formation and large scale analyses on the collected data.

Current Practices and Shortcomings: Current citizen science practices primarily rely on *passive form of crowdsourcing*, i.e., forming human networks in a rather organic way. Naturally, this form of passive crowdsourcing leads to high latency, inaccuracy, with the potential of substantial noise in the overall outcome. We, on the other hand, propose `ECCO` to enable *active crowdsourcing* for such scientific data collection, where group formation is optimization guided and the framework constantly learns about the workers from the tasks they undertake and reuse this learning. We provide one such specific scenario next in the context of ecological data collection.

EXAMPLE 1. **Scientific Data Collection and Analyses - A Citizen Science application:** *Typical citizen science efforts take place in groups to reduce errors in observation, or even keep the citizen scientists vested and motivated in the task. Imagine a citizen scientist application needs to be designed to collect data that enables building accurate predictive models by correlating environmental covariates (e.g., elevation, soil type, and average precipitation) with the presence of a species. To formulate the predictive model, the following tasks are to be performed:*

- *Confirmed Absence of Species (subtask-1): Another group (sub-group 1) needs to be formed to confirm the absence of a species. This step is considered difficult and expert workers are required to be involved to carry out this step successfully.*
- *Confirmed Presence of Species (subtask-2): A third worker group (sub-group 2) is to be created to confirm the presence of a species.*
- *Co-variate Validation (subtask-3): A fourth group of workers (sub-group 3) is created to validate the model covariates (e.g., is the elevation really 100 m at this location like the current covariate dataset indicates?).*
- *Model Validation (subtask-4): A final group of workers (sub-group 4) is tasked to validate the model itself (e.g., the system recommends a particular location for sampling a species presence/absence and the group of workers are dedicated to validate that).*
- *This iterative process terminates when the resultant data has surpassed a certain benchmark in quality (for example, the built statistical model needs to reach 90% accuracy and 85% precision). The objective is to achieve the quality as quickly as possible, by spending the smallest cost.*

Group Interactions:

- *Intra-group: Workers in the same sub-group need to interact with each other to ensure that the collected observations are correct and consistent.*
- *Inter-group: The users in sub-groups 1 and 2 are required to interact with each other to confirm the absence and presence of the species.*

Workers' Skills: *Volunteers are likely to have multiple skills, e.g., skills in ecological assessment, field training, etc.*

We attempt to abstract the processes that are likely to take place in such active crowdsourcing application and formalize them and propose `ECCO` to achieve the desired outcome. We identify the following *core aspects to support such ecological applications.*

- Complex Tasks: A citizen science application such as the one above is an example of a complex task that is composed of sub-tasks. For example, each of the data collection step described above is a sub-task and the overall task is a composition of these steps in an appropriate sequence. The current practice is to identify these sub-tasks manually [6, 12]. Interestingly, while the overall goal of a complex task may be to surpass the quality benchmarks as quickly as possible, as stated in the example, each sub-task may have different goal(s). The overall execution flow is presented in Figure 2.
- Groups: Central to such collaborative human-intensive application is the notion of "group" which may further be decomposed into sub-groups, where a set of workers collaborate with each other to complete *tasks*. Example 1 requires 4 such sub-groups.
- Human Factors: A variety of individual and group based human characteristics are to be understood [8, 9]. For example, skill of the workers to identify experts, their incentives, motivation, or ability to collaborate with each other. Some relevant skills pertinent to the running example may be ecological assessment skill, field training, and so on.
- Primary Functionalities: (a) given a complex task and a worker pool, form group of sub-groups to assign to the sub-tasks; depending on the nature of the applications, a sub-group may undertake one or more subtasks, collaborate or compete with other sub-groups. (b) learn skills and other human factors of the workers that are either individual or group based.
- Scale: We envision the necessity for a generic system that can handle a wide variety of such applications. In such a system, hundreds and thousands of citizen science workers and tasks needs to be processed and assigned. Scalable solution design becomes the first class citizen in such settings.

Desirable characteristics of ECCO**:** Several key aspects are to be appropriately unfolded: (1) uncertainty in human characteristics, namely **human factors** [8, 9] are to be understood. More importantly, we need to identify both individual human factors and those that impact group dynamism; For example, individual human factors, such as a user skill may impact how much leadership she has in a group; (2) **designing declarative primitives** which allow domain experts to easily accomplish the required functionalities; (3) **designing relevant mathematical models** to capture the appropriate optimization objectives; (4) **designing appropriate algorithms and data management techniques** for effective task assignment and human factors learning. (5) finally, **developing actual systems** or platforms that can integrate the components of ECCO.

Proposed components: ECCO consists of two primary components which are both required to be adaptive.

- *Collaborative Human Factors Learning Component:* This component first formalizes human factors - some of these characterize individuals, such as, their respective skills in different domains, their incentives (e.g., wage), motivation, as well as describe group characteristics, affinity between the workers, trust, leadership, or even application characteristics, such as, critical mass (a socio-psychological concept that describes how large a group can be for effective collaboration) [5]; These factors are then leveraged within this component to accomplish different learning, as described in Section 2. This module also exploits a "feedback" loop to enrich its learning by ingesting data coming from the deployment platform. This very aspect of users evaluating other users is a clear departure of ECCO from any existing system.
- *Human-Centric Optimization Component:* This component consists of two different optimization engines and heavily interacts with the human factors learning module to appropriately incorporate human factors in the design. (a) *Adaptive Task-Assignment Engine* is in charge of building a set of *homophilous, diverse, or complementary* groups by enabling different *interaction patterns among group members and accounting for appropriate human factors* to optimize certain outcomes of a given task. Furthermore, as stated in the running example, different groups may have different interaction pattern with each other. (b) *Human Data Management Engine*, on the other hand, manages the data learned from human factors learning component, as well as the data generated by the task assignment engine. The overall objective of this engine is to store, index, and effectively retrieve the collected data over the time. (c) Last, but not least, we wish to support *adaptivity and incremental* computation in both those engines, as human factors change over time. Figure 1 describes a high level architecture and interactivity among the different components inside ECCO.

Team formation [1] in online social networks has been the subject of some recent works which bears resemblance to the task-assignment problem. What differentiates us, is the *time-variance property and significant interoperability between different components, by deliberately acknowledging a wide variety of human factors.* Obviously, no attempt has been made to incorporate human factors learning for collaborative applications, or to effectively manage data generated by human workers.

Sections 2 and 3 contain further details. Our goal is to ensure scalability, as well as allow incremental and adaptive learning and computation. In addition to benefiting ecological and environmental science, we envision that ECCO will transpire many data management, index design, algorithmic, machine learning, and social science research problems and foster synergy across these disciplines.

2. ECCO

Individual users and applications which consist of tasks are integral part of ECCO. ECCO works in conjunction with an evaluation environment where tasks get evaluated by a human machine computation model. ECCO's components are:

2.1 Collaborative Human Factors Learning

Different collaborative applications rely on capturing and including individual human factors such as skill, motivation, acceptance ratio (describing how likely an individual will contribute) [9], or expected wage. Similarly, group human factors, such as, affinity, leadership, influence, group size (referred to as critical mass [5]) are also to be factored in. Moreover, while new users may join, existing ones may leave. Interestingly, human factors are dynamic - i.e, they change over time, and depend on the context. Human factors are also *correlated* (e.g., a highly skilled individual may be more influential, or higher rewards lead to higher acceptance ratio), and sometimes probabilistic (e.g., acceptance ratio).

While prior work [9] has acknowledged human factors, no further attempt has been made to *learn and incorporate* them in human-intensive applications in a principled manner. On the contrary, the collaborative human factors learning component is considered as one of the most fundamental contributions of ECCO, designed with the overall objective to learn the collaborative human factors [8, 9] adaptively. It proposes a set of declarative primitives to learn the (1) individual human factors, (2) group based human factors, (3) correlation among different human-factors, (4) most importantly adaptive and incremental learning of these factors, considering the achieved quality of the group based tasks. Recall the feedback loop in Figure 1 that comes from external evaluation to this component. For our citizen science example, the evaluation is performed with a hybrid human and machine intelligence. In particular, the evaluation of the completed tasks could be *precision, recall, accuracy, sensitivity*, etc. The corresponding vector in *evalmatrix* may look like, $precision = 0.8, recall = 0.6, accuracy = 0.5, specificity = 0.5$. Function `relearn` is designed to relearn how to obtain the skills of the workers (ecological assessment knowledge, field training, etc) from these evaluation values. Some example primitives are provided in Table 1.

2.2 Human-Centric Optimization

This component consists of two optimization engines.

Adaptive Task-Assignment Engine: Inputs to this engine are the user population and the tasks (or a set of subtasks), and the output are the *groups that are best suited to undertake the tasks.* Primitive `Form-Grp(`t,$\mathcal{U}$`)` is designed for this purpose, which is further explained in Table 1. No-

Primitive	Description
`human-factor-ind(u)`	output the individual human factors of user u.
`human-factor-grp(g)`	output the group based human factors of g.
`cor-human-factor(`$\{X\}, k$`)`	output the correlation among the human factors in set $\{X\}$.
`relearn(`$\{X\}', T, evalmatrix$`)`	relearn the human factors in the set $\{X\}'$, considering T tasks and the *evalmatrix*.
`Form-Grp(`t,$\mathcal{U}$`)`	output the assignment of a set of workers from the available worker pool $\mathcal{U}$ to task t.
`Form-group(L, clique,` th,$\mathcal{U}$`)`	output a clique of L workers whose aggregated all-pair affinity is th or beyond.
`add-worker(u)`, `delete-worker(u)`	adds and deletes worker u to/from the available tasks.
`find-worker(human-factor)`	find worker with a given human factor.
`find-top-worker(s)`	Find highest skilled worker for skill s.
`find-top-worker(s,k,time-period)`	Find k highest skilled worker for skill s for a given time period.
`add-eval(`g, t^1`)`	add evaluation score of subtask-1 of task t completed by group g.

Table 1: Example Primitives Descriptions Supported by ECCO.

tice that, the group formation problem for us, is *optimization mediated, instead of organic.* Needless to say that each user is described by a set of human factors that are learned from the *collaborative human factors learning* component. The criteria of the *best outcome* (i.e., optimization objectives) is domain-specific to say the least, and to be left for the domain experts to decide. We, however, provide the mechanism to incorporate these criteria into a set of well formulated optimization models.

As a simple example, given a sub-task, such as finding the workers group that can collect initial data to build the statistical model (subtask-1), the group should be formed such that the users *collectively* have the expertise to collect both positive and negative labels for the statistical model, *their wages do not surpass the cost budget of the task*, and group should be designed such that it brings forth the *maximum collaborative synergy.* On the contrary, given a complex task with a set of sub-tasks (such as one described in the running example) and described in Figure 2, we need to form a group of groups, where workers inside the same sub-group must be highly collaborative, and workers across some sub-groups need to interact with each other as well (for example, sub-group-1 and sub-group-2 in the running example). This engine is responsible to analyze the desiderata of task-assignment and form groups to enable the desirable outcome.

Naturally, even the simplest settings for such problems give rise to complex mathematical formulations having multiple-objectives to optimize. Then, the interaction pattern gives rise to constructing graphs involving the workers, where the topology of the graph should conform a specific interaction pattern, as described by the domain experts. For the example task stated in the running example, each group interaction translates to forming a clique to execute a sub-task and the interaction between the sub-groups give rise to forming a connected topology among the cliques with highest affinity, as described by Figure 2.

Finally, how big a subgroup should be is application-specific many times. We envision that ECCO would support a variety of such applications, where the group formation is premeditated and guided by a well-defined optimization objective.

Human Data Management Engine: The human factors learning component constantly generates data involving individual human workers and group of workers over the time that the task assignment engine needs to tap into to enable effective assignment of worker groups to the sub-

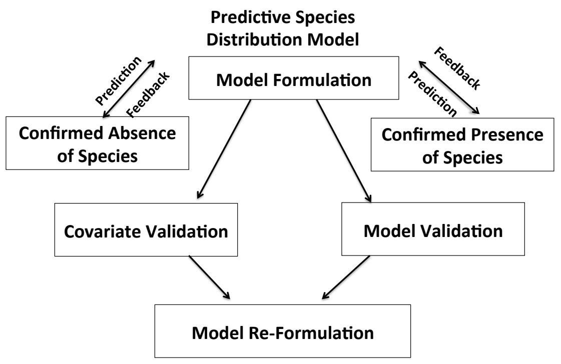


Figure 2: A Complex Task with Sub-tasks Using Example 1.

tasks. Not only that, the evaluation component generates the evaluation of the completed tasks. Interestingly, this data is temporal [3] and is associated with time stamps. Human Data Management engine provides effective management over this dataset to enable effective storage and retrieval over the time. We explain some of such functionalities next.

This engine will be enabled with the traditional database primitives, such as, add, delete, or find a user with a given human factor value, as explained in Table 1. Additionally, the domain experts or the data analysts may be interested to perform simple statistical analyses on this collected data, for example, finding the highest skill worker for a given skill s, or finding the top-k highest skilled workers, and so on. The corresponding primitives are described in Table 1. In an earlier work, we have proposed an effective indexing technique to cluster the workers based on skills and wages [9].

Recall Example 1 and notice that the complex tasks consists of a set of sub-tasks. We propose primitives to add evaluation score of a completed task (e.g., sub-task-1 in Example 1), completed by a group. Similarly, for efficient task assignment, ECCO will leverage this engine to quickly find a group of workers who are most skilled to undertake a given task (e.g., finding the best set of workers for each of the sub-tasks in Example 1). In an recent work, we propose the notion of `virtual worker`, an effective indexing technique to cluster the workers based on skills and wages for effective worker to task assignment [9]. Additionally, our proposed human data management engine is empowered to retrieve worker groups that will optimize a particular interaction pattern. For example, we will design primitives to retrieve a clique of L workers with affinity more than some threshold th. We refer back to the Table 1 again for the exact definition of the primitives.

3. CHALLENGES & DIRECTIONS

We describe some of the major challenges in realizing ECCO and our proposed directions.

(a) Identifying Relevant Factors: One significant challenge is to identify a wide variety of human factors and other necessary semantics that are needed in such applications. We consider a platform with a set of n workers, $\mathcal{U} = \{u_1, u_2, \ldots, u_n\}$[4] and l tasks, $\mathcal{T} = \{t_1, t_2, \ldots, t_l\}$. A task may be *indivisible* or may be decomposed to a set of sub-tasks. A task requires multiple skills over m different domains, $\mathcal{S} = \{s_1, s_2, \ldots, s_m\}$.
(1) Human Factors: Our initial effort has identified the following factors that potentially have a dramatic impact on the ecosystem. *(a) Skills:* The skill of a worker u is expressed as a vector $s_u^1, s_u^2, \ldots, s_u^m$ over $\mathcal{S}$, where each skill is quantified in a continuous scale between $[0, 1]$, where a value of 0 reflects no expertise for that skill. *(b) Wage/Cost:* For many collaborative tasks, explicit monetary remuneration may need to be offered to the workers. w_u represents the amount of money a worker u is willing to accept to complete a task. *(c) Acceptance Ratio:* Acceptance ratio $p_u \in [0, 1]$ of a worker u is the probability at which u accepts a task. *(d) Worker affinity:* A key to successful collaboration is the affinity among the individuals. At the atomic level, affinity is defined between a pair of workers u, u', i.e., $Aff(u, u')$ denotes how well these pair of workers work with each other.
(2) Task Quality Metrics: A task t may consist of a set $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_r\}$ of quality metrics (as described in the running example, such as precision, recall, etc). We envision that the relevant metrics to estimate the task quality would be domain specific and the *value* of a quality metric may be an *additive, multiplicative, or a more complex function* of individual worker's skills or other human factors.
(3) Constraints in Collaborative Tasks : A task t may have a budget (cost) threshold of C_t. To be considered successful, it may also have a set of quality metrics threshold Q_t^i ($i \in \{1 \ldots r\}$) with the total expenses less than C_t. Global constraints should also be considered. For example, each workers should neither be under nor be over-utilized, by assigning a lower (X_l) and an upper (X_h) limit on the number of tasks she can be assigned to.

(b) Modeling: Appropriate incorporation of the human factors is one of the foundational steps in the successful development of the *Collaborative Human Factors Learning Component*. Similarly, the *Human-Centric Optimization Component* have to formalize complex optimization problems with multiple objectives and constraints. In our initial direction, we realize that it is only realistic to collaborate with the domain experts to understand and appropriately incorporate the application specific human factors. As an example, for the species data collection task described in the running example, human factors could be ecological assessment ability, field training, workers' affinity with each other, explicit monetary incentives, etc. After that, a mathematical model is to be formalized that incorporates those factors appropriately, where it would maximize some of the factors, and use the rest as constraints. A simple mathematical model may intend to form a group $\mathcal{G}$ which maximizes the aggregated expertise Σu_{d_i} of the users as well as their collaboration affinity $\Sigma \text{Aff}(u_i, u_j)$, while keeping the total cost under a certain threshold $\Sigma w_u \leq C$, such as:

$$\text{Maximize } \Sigma_{\forall_{u_i, u_j \in \mathcal{G}}} \text{Aff}(u_i, u_j) +$$
$$\Sigma_{\forall_{u_i \in \mathcal{G}}} u_{d_i}, \Sigma_{\forall_{u \in \mathcal{G}}} w_u \leq C$$

On the contrary, if the task-assignment optimization is performed globally, we also need to add the load balancing constraints. However, the question remains, how to acknowledge in the modeling that not all users will perform according to their expertise, or a group may have to be partitioned into sub-groups if it violates the critical mass constraint. Similarly, *task creation engine* is also designed to optimize outcomes (such as minimizing latency), while satisfying the constraints provided by the domain experts.

(c) Learning: The *collaborative human factor learning* component hinges on automated learning techniques to uncover the correlation among the human factors. Several interesting and challenging problems surface that involve designing learning algorithms. For example, what makes individuals or a group remain motivated, or how to learn worker skills for collaborative tasks? Such problems are likely to give rise to novel supervised or unsupervised machine learning solutions.

Let us consider a simple illustration of the function `relearn`$(\{X\}', T, evalmatrix)$ in Section 2.1, where we are given a matrix *evalmatrix* that provides how each task t_i is evaluated based on various task quality metrics. Learning worker skills could be posed as a matrix tri-factorization problem, where *evalmatrix* is factorized as, - i.e. $evalmatrix \approx FX'G^T$ where the approximation accuracy is measured based on the norm $||evalmatrix - FX'G^T||$. Matrix $\overline{F}$ is a Boolean matrix and has the assignment of workers to groups in different tasks. X' denotes the worker to human factor matrix. The final matrix G measures the impact of human factors to task quality metrics, specifying that metric G_i as a linear combination of human factors. This tri-factorization [7] is heavily constrained using non-negativity, sparsity, row/column stochasticity, or other *marginal* constraints.

(d) Adaptivity and Incrementality: Adapativity is essential for the survival of ECCO from several perspectives - with changing time and context, individual and group human factors, as well as their correlation will vary. This not only requires the two of the first three challenges (i.e., modeling and learning) to be time-aware, but also to be adaptive in nature. For the *Human Factors Learning* component, this means that ECCO should be able to adaptively learn the human factors, as they perform more actions in the system. For the *task assignment engine*, it would mean that the system would be able to incrementally form groups as more users join or existing ones leave the platform. To enable adaptive and incremental computation the *human data management engine* needs to be sensitive to the footprints of a workers' activity in a temporal fashion. Understandably, incremental computation may introduce approximations in the results. In a recent work of ours, we have proposed how to perform adaptive task assignment by marginally solving the problem

[4]Although new workers could join and existing ones could leave any time.

and our experimental results demonstrate that our proposed solutions are both effective as well as efficient [9]. An interesting study would be to investigate the approximation factors of the proposed algorithms theoretically.

(e) Scalability: The task assignment problem is shown to be NP-hard[2], even in the simplest scenarios [9] even without considering affinity. Similarly, matrix factorization problem is inherently NP-hard [7]. Therefore, all the components of `ECCO` must ensure efficient algorithms for human factors learning or task assignment. Proposed human data management engine therefore needs to be appropriately designed to ensure efficient computations. When affinity is considered in the modeling, the very simple task assignment formulation described itself gives rise to complex graph partitioning formulation. At the same time, we intend to stay as principled as possible. Traditional modeling and learning algorithms that are typically inefficient may come largely inappropriate in our settings, thereby requiring to generate new theory and techniques. We foresee that the scalability challenges of `ECCO` will nurture engagement and collaboration across the theory, database, and machine learning community. Efficient approximation algorithms with theoretical guarantees will be proposed, or we expect to see innovative pre-computation or index design solutions to enable real time response. A very few existing research efforts [9, 1] superficially investigates some of these scalability issues for the task assignment problem. How to design the human data management engine effectively to enable efficient task assignment and human factors learning remains to be an open problem.

(f) Platform Design: `ECCO` would not be possible without the ability to conduct comprehensive experiments and validate the outcomes. Note that, finding appropriate datasets that represent the real world is one of the toughest barriers that we yet have to surpass. Most of the existing platforms, commercial or academic, such as Amazon Mechanical Turk (www.mturk.com), CrowdDB, Qurk, Deco, do not naturally support collaborative tasks. To go beyond theoretical analyses, the community needs to have access to one or more platforms that support collaboration and group formation, where the experiments and analyses can be conducted systematically. We expect that `ECCO` will transpire enough system research to build platforms and propose declarative languages to support collaborative human-intensive applications. Without appropriate evaluation strategies, indeed, the effectiveness of `ECCO` will only be partially explored.

4. CONCLUSION

We propose the vision of `ECCO`, a framework that supports data management and analyses for ecological data by leveraging the innate characteristics of individuals. We outline the two core components of `ECCO` - 1) Collaborative Human Factors Learning Component, 2) Human-Centric Optimization Component. The first component is designed to learn and characterize individual and group behaviors over time, their interdependence, which is designed to closely work with the evaluation or the deployment environment. The latter is an optimization component which interacts with the former to leverage human factors in the modeling and computations. This component is intended to automate *worker to task assignment* which are largely manual (or self meditated) and painfully slow till date. `ECCO` warrants adaptivity and scalability - to support that we propose the necessity to design an appropriate *human data management engine* that will collect and manage data coming from the human factors learning component and use that in task assignment. We intend to design principled solutions that are effective as well as efficient. We summarize the challenges of `ECCO` and propose initial directions.

5. REFERENCES

[1] A. Anagnostopoulos, L. Becchetti, C. Castillo, A. Gionis, and S. Leonardi. Online team formation in social networks. In *Proceedings of the 21st international conference on World Wide Web*, pages 839–848. ACM, 2012.

[2] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1990.

[3] C. S. Jensen and R. Snodgrass. Temporal data management. *Knowledge and Data Engineering, IEEE Transactions on*, 11(1):36–44, Jan 1999.

[4] S. Kelling, J. Gerbracht, D. Fink, C. Lagoze, W. Wong, J. Yu, T. Damoulas, and C. P. Gomes. ebird: A human/computer learning network for biodiversity conservation and research. In *Proceedings of the Twenty-Fourth Conference on Innovative Applications of Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada*, 2012.

[5] R. Kenna and B. Berche. Managing research quality: critical mass and optimal academic research group size. *IMA Journal of Management Mathematics*, 23(2):195–207, 2012.

[6] A. Kittur, B. Smus, S. Khamkar, and R. E. Kraut. Crowdforge: crowdsourcing complex work. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, UIST '11, pages 43–52, New York, NY, USA, 2011. ACM.

[7] D. D. Lee and H. S. Seung. Learning the parts of objects by non-negative matrix factorization. *Nature*, 401(6755):788–791, 1999.

[8] S. B. Roy, I. Lykourentzou, S. Thirumuruganathan, S. Amer-Yahia, and G. Das. Crowds, not drones: Modeling human factors in interactive crowdsourcing. In *DBCrowd*, 2013.

[9] S. B. Roy, I. Lykourentzou, S. Thirumuruganathan, S. Amer-Yahia, and G. Das. Optimization in knowledge-intensive crowdsourcing. *CoRR*, abs/1401.1302, 2014.

[10] J. Soberón and T. Peterson. Biodiversity informatics: managing and applying primary biodiversity data. *Philosophical Transactions of the Royal Society of London. Series B: Biological Sciences*, 359(1444):689–698, 2004.

[11] B. Sullivan, W. Christopher, M. J. Iliff, M. J. Bonney, D. Fink, and S. Kelling. ebird: A citizen-based bird observation network in the biological sciences. In *Elsevier, Biological Conservation*, 2009.

[12] H. Zhang, P. André, L. B. Chilton, J. Kim, S. P. Dow, R. C. Miller, W. E. Mackay, and M. Beaudouin-Lafon. Cobi: communitysourcing large-scale conference scheduling. In *CHI'13 Extended Abstracts on Human Factors in Computing Systems*, pages 3011–3014. ACM, 2013.

ligDB—Online Query Processing Without (almost) any Storage[*]

Evica Milchevski
University of Kaiserslautern
Kaiserslautern, Germany
milchevski@cs.uni-kl.de

Sebastian Michel
University of Kaiserslautern
Kaiserslautern, Germany
smichel@cs.uni-kl.de

"By letting it go it all gets done.
The world is won by those who let it go."
(Laozi)

ABSTRACT

In the big-data era data is arriving at such a high pace and volume that data exploration and querying can only be feasible if data loading and indexing happens reasonably quick—if at all. Recent research on handling large scientific data suggests ignoring any database indexing or even data-loading processing steps but rather turns toward processing raw data as it is handed in by scientists, manually or by semi-automated means—if needed in multiple, iterative steps. In this paper, we describe the anatomy and research challenges of a system coined LIGDB[1] that is operating purely on incomplete database tables, JSON documents, or sets of SPO triplets that are being filled over time. There is no data stored per se; the only data stored is stemming from previously posed queries over the stream of arriving data; kept as long as it is used by forthcoming queries and otherwise evicted. A key point is that velocity dimension of "big data" allows queries being processed as they are posted, with higher-level queries processed on historic query results (views) and live data. Data that is not touched by any posted query is immediately discarded.

1. INTRODUCTION

The big data challenge is about making sense of large amounts of digital content, in a timely fashion, for business intelligence or other forms of knowledge-seeking tasks. Data is generated at various sites and continuously growing; for instance through crowdsourcing missing entries of a database table, by contributing facts in Wikipedia pages, or by mentioning entity-centric properties in Tweets. The common problem dimensions imposed by the properties of what is commonly referred to as "Big Data" are best sketched by the "4Vs", most prominently volume and velocity (and variety and veracity). Data too often arrives in big volumes at high bandwidth, too much to apply traditional store-first, process-later approaches. With the advance of technology the volume and velocity of data will just keep growing, requiring a drastic shift in the current ways of data storing and processing. Recent works on handling scientific data [25, 4] have already emphasized the huge overhead of (re-)indexing for one-time queries over quickly changing data and propose processing queries on raw input data, if required through multiple iterations (parsings); or using access-driven data fetching [1]. Other attempts aim at explorative data analysis, with tools guiding the querying process and supporting approximative query results with tunable time budgets [29].

LIGDB represents a radically new approach to handling the data deluge: it is designed to let go data (hence its name) that is not required by any query and only the submission of a query triggers data gathering and processing. No other data is stored, unlike earlier works, like the ones mentioned above, where the "entire" data is available and waiting to be queried (if needed). In LIGDB only the results of queries are stored/cached, hence are treated as data by subsequent queries, until replaced in the store if not used. Think of queries that aim at finding restaurant ratings/critics or the temperature in a certain city, information on who is the fiancé of Angelina Jolie, the most promising stock to invest in, or the hottest posts in Facebook. In the massive amount of digital content that is created, commented-on, or simply re-invented (i.e., replicated) at literally every second, why should we store everything that has been produced thus far, when those questions can potentially be answered on the fly. That is, it appears possible that data is arriving at such a volume and velocity that queries can actually *wait for it.*

Once fired, queries are getting filled with result tuples until a user-specific quality or response-time criteria is met and report back to users. Ultimately, queries and their results can stay in the system and form a basis of further querying; essentially forming, traditionally speaking, a purely view-based database system over an update/data stream.

The following are the signature characteristics of LIGDB:

- **No Storage:** no data is stored per-se
- **No Schema:** data arrives in form of SPO triplets or JSON documents
- **Query-based:** queries trigger data gathering to answer the query

[*]This work has been partially supported by the German Research Foundation in project MI 1794/1-1.

[1]The prefix lig in LIGDB stands for "let it go" and *to lig* can also mean *to live on others*; both meanings capturing together the two corner stones of LIGDB

10.5441/002/edbt.2015.69

- **Result Caching:** query results are cached, statically or dynamically (as views to be updated), and evicted at some point
- **Live or Historic Results**: current queries can re-use, entirely or partially, historic query results
- **Query by Example:** queries are semi-automatically formulated, through ontological concepts and the query by example paradigm. For instance by specifying previously obtained results and waiting for the updated ones.

A system such as LIGDB is ideally able to behave, from a user perspective, like a traditional data management system: it can serve ad-hoc explorative queries and analytical queries. On the other hand, it is clear that due to the forgetful data handling, it also puts natural limits on its applicability to more traditional scenarios.

In order to build such a system, various core research areas have to be integrated. In fact, there is work on almost all aspects of this system in separation. Like data integration, result caching, view maintenance, and processing queries over data streams. Old concepts like query by example are revived to allow users querying large amounts of very sparsely (if at all) described data; a general problem for which also ontologies can help. In this work, we sketch the overall idea of LIGDB, the characteristics of its core system components, application cases, and challenges.

This paper is organized as follows. In Section 2, we review recent work on processing data on raw files or in a lazy fashion along with an overview of fundamental techniques. In Section 3 the overall architecture and core components of LIGDB are presented; including a sketch of a possible implementation. In Section 4 we discuss challenges that need to be addressed. We summarize the paper with a brief conclusion in Section 5.

2. RELATED WORK

Kersten et al. [29] envision the next generation database systems as systems that shift away from the goals of completeness and correctness, aiming towards explorative and interactive data analysis. They propose several research directions: a one-minute database kernel, producing query results within a limited time; multi-scale queries—breaking the query into multiple smaller stages; post processing of the result sets; query morphing—creating variations of the issued query and finding their results as well; and providing query suggestions for more effective data exploration.

Lazy processing: Cheung et al. [15] present Sloth, a system that extends lazy evaluation with the purpose of reducing network latency in web applications. Using dynamic program analysis, they identify the queries to be issued by the application, batch them, and postpone their processing until it is absolutely necessary. Only then they execute batches of queries, thus reducing network latency. Kargin et al. [28] propose lazy ETL, a technique for extracting, transforming and loading in a data warehouse only data that is necessary to answer the issued query. Initially, only the metadata from the queries are loaded into the warehouse. When the user issues a query, the selection predicate is imposed on the metadata to decide which files need to be loaded; the data is transformed using relational views on the extracted data; and stored into the internal data structure of the warehouse.

NoDB [25, 4] avoids data indexing and instead operates on raw csv files, if required through multiple iterations. They argue that the initial indexing in traditional DBMS poses a huge overhead, too high for one-time queries over frequently changing data. However, they still assume that data is residing on disk or another storage container and waiting to be queried. Similarly, modern SQL-on-HDFS engines like Google Dremel [35] and Cloudera Impala [16] do not own data, but execute queries over files stored on a distributed file system. Abouzied et al. [1] propose to load data for processing based on queries. While the query-driven nature is similar to what we propose with LIGDB, we do not assume that data is stored anywhere and waiting to be accessed. Instead, LIGDB operates solely on live data, as handed in by scientists or as otherwise generated. When data arrives there is a one-time chance that it gets consumed—or it is eliminated otherwise.

Approximate query processing: In [3] the authors address the problem of overestimating or underestimating the error in sampling-based approximate query processing (S-AQP). They show that existing approximation methods used by S-AQP can sometimes show errors which are overestimated or underestimated. They propose a diagnosis technique to estimate the failure of error estimation for the query, while still providing an interactive mode of answering the queries. Using the diagnostics tool, the DB system switches to non-approximate methods for answering queries, when the diagnostic tool sees that the error estimation will be unreliable.

Willis et al. [32] consider partial query results from a different perspective, i.e., partial-results generation in a case of data-access failures. They provide a taxonomy of partial-results, classifying them based on the cardinality and the correctness of the partial result with respect to the true result. They further discuss how to assess the correctness and cardinality class of partial results, and whether this can be done at a finer granularity level (e.g., per row or per column).

Ge and Golab [20] propose a framework for maintaining data structures in main-memory, sliding-window data warehouses. The proposed framework aims at combining the benefits of existing sliding-window maintenance techniques, namely single data structure for all-time window partition, and one data structure per partition.

Publish/subscribe is a widely used communication paradigm for large-scale distributed systems. In the publish subscribe interaction scheme, subscribers register for an event and publishers publish events. Subscribers are asynchronously notified when an event of interest is published. There are many variants of this system. Eugster et al. [17] identify and summarize the commonalities and differences of different publish-subscribe systems. Vargas et al. [43] propose an architecture for integrating databases with a publish-subscribe system. They integrate Hermes, a publish subscribe system, with PostgreSQL, allowing users to subscribe to events denoted by changes that occur in the underlaying database

Data Streams and Continuous Queries: Data streams have emerged as an answer to applications that do not fit the traditional data model. Golab and Özsu [22] summarize data-stream management systems from different perspectives, and further identify possible research challenges. PSoup [13] is a system for streaming queries over streaming data. PSoup treats data and queries symmetrically: when new query is registered to the system, it is probed over the historical data to find possible results. Similarly, when a new streaming data tuple arrives to the system it is first probed over the pending queries; the result is materialized; and stored in the system. A result is returned only when a user requires one, and then only the results belonging to a time

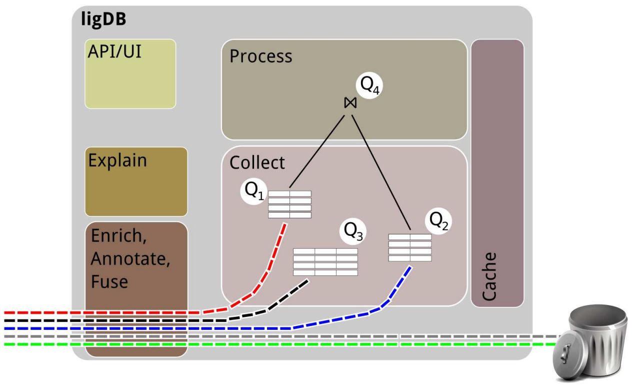


Figure 1: high level architecture

window are returned. PSoup supports joins over different data streams as well, by storing each stream in a corresponding data SteM. Bonet et al. [11, 12] discuss streaming data and the kinds of queries that can be issued. They describe types of queries over streaming data—snapshot and long-running queries. Snapshot queries are defined as those over a set of streaming sources but in a single point in time in the past. On the other hand, they describe long-running queries as queries that continually return answers as new data arrives. LIGDB is not meant to be a data-stream processing system, rather it is supposed to act like a non-streaming, traditional database system, from a user perspective; with the difference that no data is stored and only query results are cached (treated as data or, better, materialized views) and can be queried as normal "data". Terry et al. [42] propose the concept of continuous queries; a permanent query for which the user gets results whenever there is a matching tuple. They further define monotone continuous queries as queries which result is strictly non decreasing over time.

3. LIGDB ARCHITECTURE

The high-level architecture of LIGDB is shown in Figure 1. The system smoothly blends data processing on raw input streams with traditional query processing on top of data gathered in a query-driven way. We believe that the query-driven data gathering and subsequent processing in LIGDB is very reasonable as data is not created by a "big bang" but is being built up over time, for instance, crowdsourced [19], measured by sensors, or created by user actions in social networks or Wikipedia.

3.1 Core Components

LIGDB has the following core components (cf., Figure 1):

- **Enrich, Annotate, Fuse:** We consider small *data fragments* in form of sets of object-oriented (entity centric) key-value pairs as the generic data format. For instance, in form of JSON objects; however, ideally in form of full-fledged relational tables with clear schemas. In general, input data can be further annotated/enriched, e.g., through object/entity disambiguation, cleaned or standardized to general concepts using ontologies. This is very generic, and does not assume any fixed schema with full-fledged relational tables and foreign-key constraints to be present.
- **Collect:** As there is no data stored per se, initially queries are purely data-gathering queries, i.e., they define materialized views, until the query is answered to a satisfactory level. Subsequently, the query results are returned. Results can be statically cached or the query can remain in the system and needs to be continuously updated. This forms a data basis in the otherwise empty-storage LIGDB.
- **Cache:** The cache is responsible to handle the previously mentioned results of historic queries. It has space, time, and runtime constraints. That is, it has limited (in-memory) storage, evicts too-old-to-be-useful query results, and limits the amount of views to be continuously maintained [9, 30] and not frequently used.
- **Process:** Once queries are present in the system, either running or cached, newly posted queries can (fully or partially) reuse previous query results. When the query cannot be fully answered by historic results, the entire query or parts of it are posted in the data-collector component over the data stream.
- **Explain:** Querying heterogeneous, schema-free (or not well understood ones) data requires mechanisms that guide the query-phrasing process. We believe that the querying process should be driven by either examples and/or be guided by general-purpose and specialized ontologies (that can be uploaded in the system).
- **API/UI:** Users can assemble queries using the above Explain component and push them to the system. If they can solely be answered based on cached data, the result is returned instantaneous. Otherwise, the query is registered and necessary data is gathered. The user is notified if the query is answered to a satisfiable degree.

3.2 Object/Entity-Centric Input Fragments

As input, we specifically consider data represented as generic JSON objects, i.e., a bag of possibly nested key-value pairs. These might or might not come with globally unique identifiers that allows to gather data specifically related to one unique object. If ids are not given directly, to accumulate key-value pairs for the same object (entity), methods for determining the correct entity based on the data context are required. Consider for instance the JSON object in Figure 2 that gives details on a business in Phoenix, AZ, as given by the Yelp academic dataset [45]. Portals like Yelp that harness crowd input are an excellent example why data is not created in one time point, but evolves. For instance, business categories, here "Food" and "Grocery" might be added later on, review counts grow over time, and the field "open" can change over time, too. Other information such as "city" and "state" in this example are redundant, here with "full_address", but might be useful for querying. To extract such information and to bring their naming to a common ground is part of the *Enrich, Annotate, Fuse* component. There has been many recent works on understanding Web tables [33, 44] and on matching and disambiguating named entities [31, 24] that can be harnessed in LIGDB.

3.3 Query Types and Query Publishing

Per se, there is no restriction on the kind of queries that should be processed in LIGDB. Apparantly, however, in a scenario like the one addressed, where it is reasonable to throw away any historic, unused data, queries will likely be mostly of analytical, explorative nature.

The most basic query in LIGDB are so called data gathering queries that extract information out of the underlying data stream. Since it is not reasonable to assume a fixed schema, as described, one way to work is with examples/templates, for instance,

```
{
  “business_id”: “usAsSV36QmUej8–yvN-dg”,
  “full_address”: “845 W SouthernAve Phoenix,
                                     AZ 85041”,
  “open”: true,
  “categories”:[“Food”, “Grocery”],
  “city” :“Phoenix”,
  “review_count”: 5,
  “name”: “Food City”.
  “state”: ”AZ”,
  “type”: “business”
}
```

Figure 2: Excerpt of a JSON object of the Yelp dataset

```
{
  “business_id”: ?
  “name” :?,
  “city” :“Chicago”,
}
```

that are simple *selection&projection queries with predicates*. Users can also re-use existing query results to gather additional information, or to refresh/enrich previous results.

It is clear that the above samples and the focus on JSON in this work is not a restriction; likewise, we could also consider RDF Subject-Predicate-Object (SPO) triples and queries expressed in SPARQL, or even data integration/fusion into relational tables and querying with SQL. Still, it is hard to phrase meaningful queries. Below, we review some works on query advisors and ontologies [41, 7].

More advanced queries can be in a form of (top-k) aggregation queries [27], arbitrary join queries (particularly semi-join–based data pruning appears very useful), queries that gather/compute data statistics, such as our recent work on computing correlation values for entity or tag occurrences [5], up to “scientific queries” like interpolations that act as input to visualization, or data cleaning/predication on the moving data using methods such as Kalman filters; and any other queries that are traditionally processed over data streams such as running sums or quantiles. We do not explicitly rule out sliding windows to be used in LIGDB, but this, rather traditional and also orthogonal (to query-drive data gathering) concept, is not the focus in this proposal. Obviously, the selectivity of the query can impair the performance, and, in case of a *select **, turn LIGDB into a traditional store. It is the job of the below described query advisor to guide the user, considering the selectivity of the queries.

3.4 Query Processing

When considering query processing techniques, there are two important aspects of LIGDB that need to be considered. First, queries are not guaranteed to be executed within milliseconds as data is per se not given. That is, queries can run for seconds, which calls for grouping queries and sharing the load [21, 34, 14]. Second, the workload and the underlying database is very dynamic, the streaming and “historic” data as well as schemas are constantly changing. Thus, adaptive on-the-fly query optimization techniques need to be applied. Due to the dynamic nature of data streams, adaptive query processing has been addressed in [34, 8], but also for queries executed over raw data [4]. Adaptive indexing techniques [23, 26] should also be applied since, first of all, LIGDB has no fixed schemas, and second, the query workload is constantly changing—rendering fixed indexing schemes ineffective. Acosta et al. [2] discuss adaptive query processing with respect to SPARQL endpoints. Their main idea is adapting the query to the availability of the SPARQL endpoints, thus getting results even when some sources are not available, or parts of the query cannot be answered. In LIGDB, as there is no guarantee that the query can be answered as a whole, such adaptive techniques can be applied to gather answers for parts of the query.

3.5 Query Advisor

LIGDB aims at handling large amounts of heterogeneous content, not tailored to a specific, narrow application case with well designed and understood schemas. The trouble with this generic setup is that it is hard if not impossible to phrase meaningful queries without guidance. Thus, the query advisor is a key component of LIGDB. Statistics, constantly gathered from the incoming data stream (after the Enrich, Annotate, Fuse component), together with generic and domain specific ontologies, can serve as the base of the advisor. Blunschi et al. [10] employed ontologies to help keyword-based querying of complex-schema databases, for the case of business analysts in banking environments. They propose methods to find the most promising SQL-query candidates, based on input keywords. Clearly, in an arbitrary-data management system like LIGDB, this is of even higher importance.

Once a query base has been formed, and data is residing in the cache, collected data can trigger further query proposals. Sellam and Kersten propose a query advisor [39] that allows users to explore data and its statistics by posing queries that can be refined, gathering statistics and explanations of possible results. Another practical solution appears to be the application of the generic query-by-example paradigm [47] and reverse engineering of queries based on data samples [37, 46]; in addition to handing in object/entity centric (former) query results and waiting for LIGDB to refresh them. However, since all these existing techniques are designed over static data, and the nature of LIGDB implies that the data residing in the cache is dynamic, new challenges arise for adapting existing techniques to the constantly changing data.

3.6 Implementation

The input layer to LIGDB is formed by a cluster of machines that run systems such as Storm [40] or S4 [36] that aim at fault-tolerant realtime handling of big, high velocity streams of data; similar to what MapReduce is for batch processing. Such systems can scale to big loads of data by adding more machines for individual processing tasks running in parallel. Application developers have to *provide the implementation* of stream sources and operators, following the provided API. At that entry point, data is matched against registered queries and data not touched by at least a single query is immediately discarded. The Enrich, Annotate, Fuse and the Collect component is directly implemented in this streaming environment. Useful data and posted queries are indexed in an underlying key-value store by entity/object ids, including attributes and values. In the naive way, this would generate redundancy, there might be more advanced ways to store/index that information in an accessible way; such as by forming clusters or related information (e.g., in the sense of what is being done for RDF graphs). Ideally, though, a full-fledged database management system can be employed to harness standard SQL and the research efforts of the past decades (query optimization, query advisors, indices, etc.).

4. CHALLENGES

Arguably the most characteristic facet of LIGDB is its forgetful or simply ignorant way to handle data. Only posed queries trigger the collection of query-relevant data and, hence, enables their processing. This is in strict contrast to traditional data management that collects large amounts of data, indexes it, and is able to process ad-hoc queries very efficiently; assuming data is not changing frequently. This characteristic of LIGDB is unique and poses several important research challenges that need to be addressed.

4.1 Incomplete and Time-Varying Results

Terry et al. [42] define monotone continuous queries as queries whose result is non-decreasing at any point of time. However, they consider queries over append-only databases, meaning current tuples do not get updated or deleted. In LIGDB this is not the case: The same query issued only seconds later may return completely different results due to the constantly changing flow of data. One way to solve this issue is to define *truth over time*, meaning that results are true only for a specific time point or time frame, depending on the query. This is different to queries over time windows, as queries do not get attached a time window, but rather results. It is not clear whether old, cached information (even though updated) and new streaming data should always be mixed together, or if it is better to start an entirely new query. Consider for instance the case of query results that are, despite being kept fresh, capturing a larger time span; it might not be semantically correct (or advised) to join/merge its results with a new query that has seen only very recent data.

In LIGDB, we are also facing the problem of (almost) never having the complete query answer. With exception of some type of queries, results in LIGDB are never complete, by design of the system. For instance, a query asking for cities with hotel prices below some value cannot be complete as new hotel offers may always arrive to the system. The question is when query results are complete enough to be returned to users, considering a benefit/cost tradeoff between runtime and completeness/quality. One approach could investigate the level of convergence of the query result to a specific value or size, or if the tuples of the result become "stable" enough.

4.2 Scale Independence

Queries in LIGDB trigger the data collection for their processing and the amount of data they "subscribe" to is, thus, performance critical. In the sense of the concept of scale independence [6, 18], ideally, queries in LIGDB can be answered with consumption of a bounded number of input data, produce only bounded intermediate results and have also size-bounded output. Without additional knowledge and constraints imposed by the application logic, as in PIQL [6], this is not possible, and it is unclear to what extent this can be implemented "in the wild" over schema-free, heterogeneous data. In addition, queries in LIGDB should behave well in the time dimension, that is, the time required for gathering data to allow a satisfactory query answering should be ideally bounded, too. Knowing the rough output cardinality of a query helps in so far, as the degree of completion or the full completion is known. Still, this does not immediately tell when exactly data arrives that would be required to finalize the results. In fact, it might happen that a query never gets answered within reasonable time. This needs to be determined as soon as possible to return to the query initiator. The system should also be interactive in the sense that it periodically returns incremental results or time-to-completion information.

4.3 Cold-Start Problem

In recommender systems, a cold-start problem (cf., [38]) occurs when a new item (user) is added to the system which cannot be recommended as no one has rated the item so far. In LIGDB there is a cold-start problem when a query is posted for which only few or no data at all is present in the system, as in the recent past no related query has been issued. With changing data and user interests this problem is expected to occur virtually at any time. In that case, all query-related data need to be first gathered from scratch and no previously cached query result is useful to answer the query at least partially. After this cold-start phase it is assumed that most queries can be answered at least partially by harnessing cached query results. Partially answerable means that previous queries allow finding a subset of the true query answers (objects) with potentially missing key-values pairs in the individual result entries. It is to be decided in this case if it is reasonable to postpone the result delivery to the query initiator in order to gather additional results and attributes, to already report the incomplete results, and/or to start data gathering to aim at more complete results. Another issue which rises with the cold-start problem is how and which queries to phrase: Phrasing queries when there is no schema available or some previous knowledge of the streaming data is not trivial. This could make the user resort to posing general data-gathering queries. Thus the LIGDB query advisor, ideally, by using the gathered statistics and/or ontologies, should be able to suggest queries in this case as well.

5. CONCLUSION

In this work, we sketched the core ideas and research challenges behind LIGDB, a data-management architecture that makes the case for storing only data for which a related query is posted to the system. This is in strong contrast to common data management systems, that follow an index-first-query-later paradigm or are running continuous queries (often of statistical nature) on data streams. LIGDB is designed to ideally act as a traditional data management system, serving ad-hoc queries in acceptable response times but can also harness long-running data stream analytics. The forgetful data handling and the pay-as-you-go building up of a data repository to be reused render LIGDB appealing for handling explorative and analytical queries over large input data. The price to pay for such a data processing principle is its susceptibility to data-to-query discrepancy and a not sufficiently large data input rate.

6. REFERENCES

[1] A. Abouzied, D. J. Abadi, and A. Silberschatz. Invisible loading: access-driven data transfer from raw files into database systems. *EDBT*, 2013.

[2] M. Acosta, M.-E. Vidal, T. Lampo, J. Castillo, and E. Ruckhaus. Anapsid: An adaptive query processing engine for sparql endpoints. *ISWC*, 2011.

[3] S. Agarwal, H. Milner, A. Kleiner, A. Talwalkar, M. I. Jordan, S. Madden, B. Mozafari, and I. Stoica.

Knowing when you're wrong: building fast and reliable approximate query processing systems. *SIGMOD*, 2014.

[4] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. Nodb: efficient query execution on raw data files. *SIGMOD*, pages 241–252, 2012.

[5] F. Alvanaki and S. Michel. Tracking set correlations at large scale. In *SIGMOD*, 2014.

[6] M. Armbrust, K. Curtis, T. Kraska, A. Fox, M. J. Franklin, and D. A. Patterson. Piql: Success-tolerant query processing in the cloud. *PVLDB*, 5(3), 2011.

[7] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. G. Ives. Dbpedia: A nucleus for a web of open data. *ISWC/ASWC*, 2007.

[8] S. Babu and J. Widom. Streamon: An adaptive engine for stream query processing. *SIGMOD*, 2004.

[9] J. A. Blakeley, P.-Å. Larson, and F. W. Tompa. Efficiently updating materialized views. *SIGMOD*, 1986.

[10] L. Blunschi, C. Jossen, D. Kossmann, M. Mori, and K. Stockinger. Soda: Generating sql for business users. *PVLDB*, 5(10), 2012.

[11] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. *MDM*, 2001.

[12] P. Bonnet and P. Seshadri. Device database systems. *ICDE*, 2000.

[13] S. Chandrasekaran and M. J. Franklin. Psoup: A system for streaming queries over streaming data. *The VLDB Journal*, 12(2), 2003.

[14] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. *SIGMOD*, 2000.

[15] A. Cheung, S. Madden, and A. Solar-Lezama. Sloth: being lazy is a virtue (when issuing database queries). *SIGMOD*, 2014.

[16] Cloudera Impala. `https://github.com/cloudera/impala`.

[17] P. T. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2), 2003.

[18] W. Fan, F. Geerts, and L. Libkin. On scale independence for querying big data. *PODS*, 2014.

[19] A. Feng, M. J. Franklin, D. Kossmann, T. Kraska, S. Madden, S. Ramesh, A. Wang, and R. Xin. Crowddb: Query processing with the vldb crowd. *PVLDB*, 4(12), 2011.

[20] C. Ge and L. Golab. Lazy data structure maintenance for main-memory analytics over sliding windows. *DOLAP*, 2013.

[21] G. Giannikis, D. Makreshanski, G. Alonso, and D. Kossmann. Shared workload optimization. *PVLDB*, 7(6), 2014.

[22] L. Golab and M. T. Özsu. Issues in data stream management. *SIGMOD Record*, 32(2), 2003.

[23] G. Graefe and H. A. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. *EDBT*, 2010.

[24] J. Hoffart, Y. Altun, and G. Weikum. Discovering emerging entities with ambiguous names. *WWW*, 2014.

[25] S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. Here are my data files. here are my queries. where are my results? *CIDR*, 2011.

[26] S. Idreos, S. Manegold, H. A. Kuno, and G. Graefe. Merging what's cracked, cracking what's merged: Adaptive indexing in main-memory column-stores. *PVLDB*, 4(9), 2011.

[27] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.

[28] Y. Kargín, M. Ivanova, Y. Zhang, S. Manegold, and M. Kersten. Lazy etl in action: Etl technology dates scientific data. *PVLDB*, 6(12), 2013.

[29] M. L. Kersten, S. Idreos, S. Manegold, and E. Liarou. The researcher's guide to the data deluge: Querying a scientific database in just a few seconds. *PVLDB*, 4(12), 2011.

[30] C. Koch, Y. Ahmad, O. Kennedy, M. Nikolic, A. Nötzli, D. Lupei, and A. Shaikhha. Dbtoaster: higher-order delta processing for dynamic, frequently fresh views. *VLDB J.*, 23(2), 2014.

[31] H. Köpcke, A. Thor, and E. Rahm. Evaluation of entity resolution approaches on real-world match problems. *PVLDB*, 3(1), 2010.

[32] W. Lang, R. V. Nehme, E. Robinson, and J. F. Naughton. Partial results in database systems. *SIGMOD*, 2014.

[33] G. Limaye, S. Sarawagi, and S. Chakrabarti. Annotating and searching web tables using entities, types and relationships. *PVLDB*, 3(1), 2010.

[34] S. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. *SIGMOD*, 2002.

[35] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. *Commun. ACM*, 54(6), 2011.

[36] S4: Distributed stream computing platform. `http://incubator.apache.org/s4/`.

[37] A. D. Sarma, A. G. Parameswaran, H. Garcia-Molina, and J. Widom. Synthesizing view definitions from data. *ICDT*, 2010.

[38] A. I. Schein, A. Popescul, L. H. Ungar, and D. M. Pennock. Methods and metrics for cold-start recommendations. *SIGIR*, 2002.

[39] T. Sellam and M. L. Kersten. Meet charles, big data query advisor. *CIDR*, 2013.

[40] Storm: Distributed and fault-tolerant realtime computation. `http://storm-project.net/`.

[41] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: a core of semantic knowledge. *WWW*, 2007.

[42] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. *SIGMOD*, 1992.

[43] L. Vargas, J. Bacon, and K. Moody. Integrating databases with publish/subscribe. *DEBS*, 2005.

[44] P. Venetis, A. Y. Halevy, J. Madhavan, M. Pasca, W. Shen, F. Wu, G. Miao, and C. Wu. Recovering semantics of tables on the web. *PVLDB*, 4(9), 2011.

[45] Yelp Academic Dataset. `https://www.yelp.com/academic_dataset`.

[46] M. Zhang, H. Elmeleegy, C. M. Procopiuc, and D. Srivastava. Reverse engineering complex join queries. *SIGMOD*, 2013.

[47] M. M. Zloof. Query-by-example: the invocation and definition of tables and forms. *VLDB*, 1975.

Procrastination Beats Prevention

Timely Sufficient Persistence for Efficient Crash Resilience

Faisal Nawab[*,†] Dhruva R. Chakrabarti[†]

Terence Kelly[†] Charles B. Morrey III[†]

[*]CS Dept., UC Santa Barbara [†]HP Labs, Palo Alto, CA

ABSTRACT

Preserving the integrity of application data across updates in the presence of failure is an essential function of computing systems, and byte-addressable non-volatile memory (NVM) broadens the range of fault-tolerance strategies that implement it. NVM invites database systems to manipulate durable data directly via LOAD and STORE instructions, but overheads due to the widely used mechanisms that ensure consistent recovery from failures impair performance, e.g., the logging overheads of transactions. We introduce the concept of *Timely Sufficient Persistence* (TSP) mechanisms, which is relevant to both conventional and emerging computer architectures. For a broad spectrum of fault-tolerance requirements, satisfactory TSP mechanisms typically involve lower overheads during failure-free operation than their non-TSP counterparts; hardware and OS support can facilitate TSP mechanisms. We present TSP variants of programs representing two very different classes of shared-memory multi-threaded software that store application data in persistent heaps: The first employs conventional mutexes for isolation, and TSP substantially reduces the overhead of a fault-tolerance mechanism based on fine-grained logging. The second class of software employs lock-free and wait-free algorithms; remarkably, TSP is very easy to retrofit onto a non-resilient design and enjoys *zero runtime overhead*. Extensive experiments confirm that TSP yields robust crash resilience with substantially reduced overhead.

1. INTRODUCTION

Runtime failures such as process crashes, operating system kernel panics, and power outages can corrupt or destroy application data unless effective measures protect application data integrity. Both disk-based and main-memory database systems running on conventional hardware with volatile byte-addressed memory and non-volatile block storage employ sophisticated techniques to preserve the integrity of application data across updates in the presence of failures [13]. Unfortunately these techniques sometimes suffer painful runtime overheads due to the performance characteristics of hard drives and solid state drives.

Emerging hardware promises durability with greatly improved performance [14]. Byte-addressable non-volatile memory (NVM) is becoming available [1] and has enabled new database system designs with improved performance [17, 21, 26]. NVM has sparked increased interest in main-memory databases that store all data in memory and directly manipulate durable data in persistent heaps via LOAD and STORE instructions rather than through database or filesystem interfaces [25]. Such approaches offer superior performance compared to disk- and SSD-based systems with comparable fault tolerance, but they suffer noticeable overheads during failure-free operation [14].

We begin by introducing a conceptual framework that encompasses both conventional and emerging hardware. We then systematically characterize as a function of fault-tolerance requirements the circumstances under which runtime overheads may safely be postponed until failures actually occur and/or eliminated outright, and we tailor the specific measures taken to the available hardware. The result is the concept of *Timely Sufficient Persistence* (TSP). Loosely speaking, a TSP fault-tolerance mechanism eschews costly preventive measures in favor of minimalist remediation when failure is imminent, which typically reduces runtime overheads substantially. It furthermore helps us to identify new hardware and OS support to facilitate new fault tolerance mechanisms. We restrict ourselves in this paper to the context of a single computer, and the following types of failures: process crashes, kernel panics, and power outages.

The remainder of this paper is organized as follows: Section 2 reviews emerging hardware architectures, describes corresponding fault tolerance mechanisms and application programming styles, and discusses how these new developments relate to traditional fault-tolerance objectives. Section 3 defines Timely Sufficient Persistence and describes how it can be implemented on both conventional and emerging hardware. Section 4 presents two case studies illustrating the benefits of TSP: In one case, TSP imbues a program with crash resilience while adding *zero* runtime overhead; in another case, TSP substantially reduces the runtime overhead of an existing fault-tolerance technique. Section 5 presents experimental results confirming both the fault tol-

10.5441/002/edbt.2015.70

erance properties and performance advantages of TSP in our two case studies, and Section 6 concludes with a discussion.

2. NEW HARDWARE & SOFTWARE

Database management systems are designed to survive severe failures, e.g., power outages. When they run on conventional hardware with volatile DRAM memory, they must therefore write data to block-addressed storage devices, which perform poorly for random writes. Write-ahead logging and grouping partially mitigate the storage I/O bottleneck [13]. The need to synchronously commit such writes to block storage may limit overall database performance to storage bandwidth [10]. Solid state drives (SSDs) enable databases with improved performance compared with disk-based designs [19]. However, SSDs still suffer from the same fundamental drawback as HDDs: both kinds of storage devices require databases to synchronously commit data via a relatively slow block I/O interface [19]. Main-memory databases are optimized for the case where all data fits in memory [22]. To this end, systems such as HBase [9] and Silo [24] redesign recovery and locking mechanisms so as to minimize the impact of these bottlenecks. However, main-memory databases that have not been re-architected for the case that all data fits in durable memory still suffer from the I/O bottleneck of persisting on stable storage. Traditional filesystems running on conventional hardware provide an alternative means of manipulating durable data, but they suffer the same storage bottlenecks that afflict databases [4].

Byte-addressable non-volatile memory (NVM) has recently become available. Conventional DRAM is approaching density scaling limits [1], and the most promising replacement technologies—phase change memory, spin-torque transfer memory, and memristors—all happen to be non-volatile [26]. If any of these technologies architecturally supplants or supplements DRAM it will provide inherently non-volatile random-access memory (NVRAM). Meanwhile, hybrid DRAM/flash memory DIMM packages backed by batteries or supercapacitors (NVDIMMs) implement NVM by persisting the contents of DRAM to flash when power is lost [14]. Non-volatile CPU caches have been proposed to complement NVM [28]. Another way to preserve the contents of volatile DRAM across utility power outages is to fail over to an uninterruptible power supply, a traditional building block of fault-tolerant systems [12]. Regardless of the underlying technology, all forms of NVM share several advantages over block-addressed storage devices. NVM is installed on the memory bus and enjoys access latencies and bandwidth comparable to DRAM. NVM is accessed at cache-line granularity via LOAD and STORE instructions. In contrast to the relatively coarse, slow, mediated updates offered by database and file systems atop block storage devices, NVM enables fast, fine-grained, direct updates by application software.

The potential of NVM has been explored in the context of disk-based and main-memory databases [17, 21, 26] and filesystems [5]. However the arrival of new forms of NVM has also renewed interest in a style of application programming that has long been possible but that has remained outside the mainstream until recently. Since the days of MULTICS, some operating systems have offered application programs the *illusion* that LOAD and STORE instructions operate upon durable data [6]; file-backed memory mappings provide this illusion on modern POSIX systems [23]. Atop such mechanisms it is possible to layer higher-level abstractions ranging from straightforward persistent heaps [18] to sophisticated object databases [27].

Compared with the more mainstream approach in which applications manipulate durable data via filesystem or database interfaces, the "NVM style" of direct manipulation offers several attractions. The most obvious is that in-memory data structures and algorithms are sometimes more convenient and more natural than the storage-oriented alternatives. A related issue is that translating between in-memory and serial data formats can be cumbersome. Translation between serial and in-memory formats can also be slow and error-prone; parsers, for example, are notorious for harboring bugs.

Supporting fault-tolerant "NVM-style programming" in the age of genuine NVM presents interesting new opportunities and challenges. Failures that abruptly terminate program execution can leave application data in NVM in an inconsistent state, so the challenge is to ensure that recovery can always restore consistency to data that survives the failure. Recent research has proposed transactional updates of persistent heaps, where transactions are defined either explicitly by the programmer [25] or are automatically inferred from the target program's use of mutual exclusion primitives [2, 3]. In the latter approach, synchronously flushing UNDO log entries to NVM immediately before STORE instructions execute enables recovery code to roll back transactions as necessary to restore the persistent heap to a consistent state, and such synchronous flushing adds noticeable overhead during failure-free operation.

Fortunately, some characteristics of emerging hardware work to our advantage when addressing specific kinds of failures. For example, the time and energy costs of flushing volatile CPU cache contents to the safety of NVM are miniscule compared to the corresponding costs of evacuating data in volatile DRAM to block storage [14]—a crucial difference that helps enormously if we must quickly panic-halt a faulty OS kernel. In general, the key to finding the best designs for meeting given fault-tolerance requirements on emerging hardware is to systematically consider the costs of moving data out of harm's way and to devise contingency plans that replace burdensome migrations during failure-free operation with guarantees of last-minute rescue. We shall see that emerging architectures sometimes reward procrastination handsomely.

3. TIMELY SUFFICIENT PERSISTENCE

Application requirements must distinguish tolerated failures from non-tolerated failures. Process crashes and kernel panics resulting from software or hardware errors are frequently placed in the former category, as are power outages. Application requirements must furthermore specify what subset of *critical* application data must survive tolerated failures. For example, requirements might declare that the entire state of a process is critical; more selective requirements might instead deem the process heap to be critical but permit thread execution stacks to be lost. Requirements might even designate different fault tolerance requirements for different subsets of application data. Requirements must also distinguish between fail-stop failures that abruptly halt process/thread execution and failures that first corrupt application data. For example, when a process on a POSIX system receives a `SIGKILL` signal, all threads merely halt; the same is sometimes true when a process triggers a trap, e.g., by executing illegal instructions. By contrast, memory corruption errors in C/C++ programs often corrupt critical application data.

Fault-tolerance strategies typically move data from places where tolerated failures threaten corruption or destruction to places beyond the reach of tolerated failures; we respectively refer to such locations as *vulnerable* and *safe*. Safety can be defined only with respect to fault-tolerance requirements and is orthogonal to hardware characteristics such as volatility. For example, ordinary volatile DRAM can be safe with respect to process crashes, but even hard disks may be deemed vulnerable if we must tolerate catastrophes that wipe out entire data centers. Finally, we must ask whether we have adequate notice of tolerated failures to move critical data from vulnerable locations to safe ones. If so, we may seek improved performance while still meeting fault-tolerance requirements by trading runtime guarantees that critical data *is* in a safe location for guarantees that the data *will be moved to safety* should the need ever arise.

Timely Sufficient Persistence (TSP) describes fault-tolerance mechanisms that make such tradeoffs. A TSP design satisfies its requirements by moving a *minimal* amount of data (typically only critical data) to a location that is *adequately* safe (typically no safer) and does so in a *timely* manner (typically "just in time"). For example, Whole System Persistence [14] is an ingenious two-stage TSP design that protects the entire state of a computer from power outages by flushes the contents of volatile CPU registers and caches into volatile DRAM using residual energy stored in the system power supply and then evacuating the contents of DRAM into flash storage using energy stored in supercapacitors. This design completely avoids any overhead during failure-free operation. Presently we shall consider other TSP designs that tolerate a wider range of failures (e.g., due to software errors), that protect critical data more selectively, and that offer similarly attractive performance.

In our experience it is instructive to ask simple questions about the minimum support needed to satisfy given fault tolerance requirements—if only just barely—and to ask what "hidden" support may be present in the hardware and systems we are already using. Such exercises have more than once led the authors to insights that in turn informed improved TSP designs that, in retrospect, had been right under our noses but that we had overlooked before we began to seek TSP solutions explicitly. For example, consider the requirement that critical data that is explicitly placed in memory allocated through a special interface must survive process crashes only. A naïve approach might begin with the observation that physical memory allocated to a process is promptly reclaimed by the OS when the process crashes, with no opportunity for the process to rescue its contents. This line of reasoning might then conclude that crash tolerance in this context requires preemptively (and perhaps synchronously) committing data to durable media during failure-free operation.

A better approach begins by asking what minimal degree of "durability" suffices to survive process crashes: POSIX calls it "kernel persistence," and files in memory-backed filesystems have this property. We then consider what happens when such a file is memory mapped into the address space of a process that STOREs data into the mapping region and then crashes. The modified physical memory page frames corresponding to the mapping are also pages in the backing file and are *not* reclaimed by the OS when the process crashes. Furthermore STOREd data in the CPU cache at the time of the crash will eventually be evicted into the memory-backed file and meanwhile will be visible from the cache to any process that reads the file. Therefore if the process places critical data in memory corresponding to a memory-mapped file from a DRAM-backed file system, following a crash the file will contain all data STOREd by the process up to the instant of the crash, and we obtain this guarantee with no overhead during failure-free operation. (Our technical report provides additional detail and references on the interaction between process crashes and file-backed memory mappings [15].) Of course, additional measures may be required to ensure that application data stored in the file can be restored to a consistent state following a crash; we consider two different ways of ensuring consistent recoverability in Section 4. The important point is that seeking a TSP solution has gotten us halfway to our goal with zero runtime overhead.

Different kinds of failures call for different TSP designs. If we are required to tolerate kernel panics, for example, we must arrange for the dying OS to flush volatile CPU caches to memory. This suffices to meet the requirement if memory is non-volatile (or if the machine architecture preserves the contents of memory across "warm reboots" [16]). If memory is volatile and is not preserved across OS re-starts, the contents of memory must be written to stable storage before the panic'd OS shuts down the machine. An HP team has implemented the required support in the Linux kernel's panic handler, which required a relatively small amount of straightforward code. Power outages admit a spectrum of TSP designs ranging from mundane uninterruptible power supplies to sophisticated and resourceful strategies for storing and scrounging just enough energy to rescue critical data [14]. Emerging non-volatile memories can dramatically reduce the time and energy cost of keeping a machine running long enough to rescue critical data after utility power fails.

Conventional relational database management systems allow the user to trade consistency for performance via configuration parameters. For example, serializability and snapshot isolation offer different performance and consistency guarantees. TSP designs provide a wider range of applications with analogous tradeoffs among failure toleration requirements, hardware and system software support, and performance during failure-free operation.

4. CASE STUDIES

We now consider in detail two approaches to ensuring consistent recovery of application data in multi-threaded programs that manipulate persistent heaps via CPU LOAD and STORE instructions. Both approaches share several features in common: the programming model is convenient, familiar, and readily implementable in mainstream programming languages such as C++; the programmer obtains access to address space regions backed by durable media via a conventional memory allocation interface (e.g., `malloc` for C/C++); and the programmer assists recovery by ensuring that all live application data in the persistent heap are reachable from a heap-wide root pointer manipulated via simple `get_root()` and `set_root()` interfaces. Finally, in both approaches the application programmer must ensure that concurrent threads access shared data in an orderly manner, free of data races and other concurrency bugs. In one of our approaches, multithreaded isolation depends upon conventional synchronization primitives (e.g., Pthread mutexes); the other relies upon non-blocking algorithms. We describe how TSP enables both kinds of multithreaded software to ensure consistent recovery of the persistent heap without high-latency CPU cache flushing during failure-free operation.

Implementations of both approaches on emerging architectures featuring NVRAM or NVDIMM memory offer substantial advantages, but implementation on conventional hardware (volatile DRAM and block-addressed storage) is also possible. To tolerate process crashes only, it suffices to ensure that the persistent heap is backed by a memory-mapped file in an ordinary filesystem or even a file in a DRAM-backed file system (e.g, `/dev/shm`). To tolerate kernel panics, the kernel must flush volatile CPU caches to memory; if the latter is volatile, memory regions corresponding to persistent heaps must be written to durable storage. To tolerate power outages, sufficient standby power must be available to flush CPU caches and move persistent heap data to durable media. As noted in Section 1, NVRAM and NVDIMMs dramatically reduce the time and energy cost of tolerating both kernel panics and power outages.

Sections 4.1 and 4.2 describe the principles underlying the two approaches; Section 5 describes corresponding implementations and our empirical evaluation of their correctness and performance overheads.

4.1 Zero-Overhead Atomic Updates

This section presents the remarkable observation that a well-known class of multi-threaded *isolation* mechanisms, together with TSP, guarantee *consistent recovery* from crashes without the need for any additional mechanisms or precautions whatsoever.

Following the terminology of Fraser & Harris [8], we say that an algorithm that ensures orderly multi-threaded access to shared in-memory data is *non-blocking* if the suspension or termination of any subset of threads cannot prevent remaining active threads from continuing to perform correct computation. Non-blocking algorithms cannot employ conventional mutual exclusion because a mutex held by a terminated thread will never be released, which prevents all surviving threads from accessing data protected by the mutex. Threads in non-blocking algorithms typically employ atomic CPU instructions such as compare-and-swap to update shared memory while precluding the possibility that other threads may observe inconsistent states of application data. *Lock-free* algorithms, a special case of non-blocking algorithms, offer the stronger guarantee that forward progress occurs even in the presence of contention for shared data. *Wait-freedom* is yet a stronger guarantee that a bounded number of operations is needed to complete an operation. All wait-free algorithms are lock-free. We employ lock-free and wait-free sub-species of non-blocking algorithms, using the latter term for brevity. One additional definition helps us to reason about the effects of crashes: Following Pelley et al. [20], we imagine a thread called the *recovery observer* that is created at, and observes the state of program memory at, the instant when all other threads in a program abruptly halt due to a crash.

Consider a program whose application-level data resides in a persistent heap and is manipulated with a non-blocking algorithm. The heap is furthermore updated in TSP fashion, i.e., in the event of a crash due to any tolerated failure, data in volatile locations (e.g., CPU caches or DRAM) will be flushed to durable media (NVRAM/NVDIMMs or stable storage) as necessary. We shall see that under these assumptions, a crash cannot prevent consistent recovery of the application data in the persistent heap.

Consider a crash that abruptly terminates all of the program's threads. We imagine a recovery observer created at the instant of the crash and consider its view of memory. Thanks to TSP, practical/implementable recovery code will have precisely the same view of memory as our hypothetical recovery observer. In particular, TSP ensures that the state of recovered memory will reflect a strict prefix of the STORE instructions issued by the terminated threads. By definition of non-blocking algorithm, the termination of the program's threads by the crash cannot prevent the recovery observer from making correct progress based on its view of memory, regardless of what the recovery observer intends to do. In particular, the recovery observer may traverse application data in the persistent heap by starting at the heap's root pointer; again by the definition of non-blocking algorithm, the recovery observer will never thereby encounter corrupt or inconsistent application data. Identical reasoning applies to any number of recovery observers, which collectively could resume correct execution from the consistent state of application data that they find in the persistent heap.

The main advantage of the approach outlined above is that it requires relatively little additional effort for the class of software to which it applies. Unlike whole-system persistence (WSP) [14], our technique does not simply resume thread execution where a crash suspended it—which would be fine for power outages but which isn't the right remedy for crashes induced by software bugs. Instead, we require application code to resume execution from a consistent state of the persistent heap. However our technique is potentially applicable to a broader range of failures, including not only the power outages handled by WSP but also software failures including kernel panics and process crashes, so long as the failures do not corrupt the persistent heap. One restriction of the approach outlined above is the requirement that applications manipulate data in persistent heaps exclusively via non-blocking algorithms. Such algorithms may offer excellent performance, but they are less general, more complex, and less widely used than alternative approaches. We now consider how TSP enables efficient support for consistent recoverability in a much wider class of software.

4.2 Mutex-Based Software

Atlas is a system that employs compile-time analysis and instrumentation, run-time logging, and sophisticated recovery-time analysis to imbue conventional mutex-based multithreaded software with crash resilience [2,3]. Atlas operates upon multi-threaded programs that correctly employ mutexes to prevent concurrency bugs and ensure appropriate inter-thread isolation but that take no measures whatsoever to ensure consistent recovery from durable media. Atlas is nearly transparent, requiring minimal changes to target programs: durable data must reside in a persistent heap and all active data structures in the persistent heap must be reachable from the persistent heap's root pointer. Atlas guarantees that recovery will restore the persistent heap to a consistent state and that crashes cannot corrupt the integrity of data within it. We explain how TSP improves performance during failure-free operation after briefly reviewing the workings of Atlas; previous publications supply the details [2,3].

Atlas leverages the fact that shared heap data may be modified within critical sections protected by mutexes and assumes that each outermost critical section (OCS) in the target program both finds and leaves the heap in a consistent state according to application-level integrity criteria. Therefore each OCS represents a bundle of changes to the persistent heap that should be applied failure-atomically. Atlas instruments target programs with logging mechanisms to ensure that an OCS interrupted by a crash can be rolled back during recovery. Furthermore, subtle interactions among OCSes can produce situations where OCSes that completed *prior* to a crash must nonetheless be rolled back upon recovery (see Section 2.3 of [2]); Atlas recovery code correctly handles such situations. Finally, it is possible for crashes to cause Atlas-fortified software to leak memory; Atlas recently incorporated a recovery-time garbage collector to reclaim leaked memory.

Compared with the approach to consistent recovery of programs that employ non-blocking algorithms described in Section 4.1, Atlas offers several advantages: Atlas operates upon more general classes of software that employ familiar isolation mechanisms, as opposed to more restricted and much more esoteric non-blocking algorithms. Furthermore, because Atlas rolls back critical sections interrupted by crashes, it can tolerate failures that cause data corruption within such critical sections; thus Atlas-fortified software is robust against a wider range of failures.

Timely Sufficient Persistence brings substantial performance benefits to Atlas-fortified software. Atlas employs UNDO logging at run time to retain the ability to roll back OCSes during recovery: Before allowing a STORE instruction in the target program to alter a persistent heap location for the first time in an OCS, Atlas first adds an entry to its UNDO log. If TSP is not available, Atlas must *synchronously* flush the UNDO log entry from the CPU cache into memory before allowing the STORE to occur. This synchronous flushing adds considerable overhead beyond the unavoidable Atlas overhead of logging. However if TSP is available, synchronously flushing CPU caches is no longer necessary because TSP guarantees that recovery will read the most recent state of all persistent memory locations, regardless of what tolerated failure has occurred. The details of how TSP delivers on this guarantee will of course depend on the details of *how* TSP tolerates failures (Section 3).

5. EXPERIMENTS

We performed fault-injection experiments to confirm that both of the approaches described in Section 4 do indeed ensure consistent recovery of persistent heap data. We also measured the overhead of the logging required by Atlas (Section 4.2) and of the failure-free cache flushing that Atlas would require if TSP were not available. Previously published experiments applying Atlas to real applications (OpenLDAP and `memcached`) and benchmarks (Splash2) have shown a 3× performance overhead of logging alone and 5× overhead when both logging and synchronous flushing are enabled [3]. The more recent results in Section 5.2 below extend and confirm our earlier findings.

5.1 Map Interface & Implementations

Our experiments employ two different multi-threaded implementations of the familiar "map" interface, i.e., a local key-value store that in the present case maps integer keys to integer values. We divide the key space into a small lower range L used for integrity checks and the remaining much larger higher range H. Each thread $t \in [1 \dots T]$ maintains in the map two private counters indexed with keys $c_{1,t}$ and $c_{2,t}$ in L. Iteration i of the main loop of each worker thread performs three steps as atomic and isolated operations: it first sets the value associated with $c_{1,t}$ to i, then increments the value associated with a key drawn with uniform probability from H, then sets the value associated with $c_{2,t}$ to i. The correctness invariants of the map are the following two inequalities:

$$\sum_{t=1}^{T} c_{1,t} - \sum_{t=1}^{T} c_{2,t} \quad \leq \quad T \tag{1}$$

$$\sum_{t=1}^{T} c_{1,t} \quad \geq \quad \sum_{\text{key}\, k \in H} \text{map}[k].\text{value} \quad \geq \quad \sum_{t=1}^{T} c_{2,t} \tag{2}$$

Our non-blocking map implementation is based on a lock-free skip list by Herlihy & Shavit [11]. We employ a mature and stable C implementation by Dybnis that is believed to be bug-free [7]. We wrote our own mutex-based map implementation in C. It employs a separate-chaining hash table and moderate-grain locking (one mutex per 1000 buckets).

Our fault-injection methodology mimics the effects of a sudden process crash caused by an application software error, e.g., a segmentation violation, illegal instruction, or integer divide-by-zero. We abruptly and simultaneously terminate all threads in a running process by sending the process a `SIGKILL` signal, which cannot be caught or ignored. Recovery code then attempts to locate the map in the persistent heap by starting from the heap's root pointer, traverse the contents of the map, and verify the integrity of the map by testing the invariants of Equations 1 and 2.

5.2 Results

Both our map implementations recovered completely successfully after hundreds of injected process crashes, consistent with previous findings concerning Atlas [3] and with the reasoning in Section 4.1. Similar results would occur under other kinds of non-corrupting failures.

We measured the performance of four variants of our map implementations, where the metric used is "total number of iterations of all worker threads per second" (recall from Section 5.1 that each iteration performs three atomic operations). The throughput of our native unmodified mutex-based code is compared with two Atlas-fortified variants of the same code, one with UNDO logging alone and one with both logging and synchronous CPU cache flushing. We can thus quantify the overhead of logging alone, which is sufficient for consistent recovery if TSP is available, and of synchronous flushing, which is necessary for consistent recovery if TSP is not available. We include the performance of the non-blocking map for completeness, noting that com-

Hardware Platform				Throughput (millions iter/sec)			
	CPU type	hardware		Mutex-Based			
Computer	@ GHz	threads	DRAM	no Atlas	log only	log + flush	Non-Blocking
ENVY Phoenix 800 Desktop	i7-4770 @ 3.4	8	32 GB	3.66	2.36	1.58	2.54
DL580 Gen8 Server	E7-4890v2 @ 2.8	30	1.5 TB	2.13	1.50	1.06	2.00

Table 1: Hardware platforms & experimental results. All computers are HP, all CPUs Intel.

parisons with the mutex-based map are problematic because the two maps employ different data structures (hash table vs. skip list). All performance and fault-injection experiments were conducted on the HP/Intel computers described on the left-hand side of Table 1.

The right-hand side of Table 1 presents our performance results. In all cases we report results for runs with eight worker threads. For the server experiment we pinned all software threads to a single one of the DL580's four CPU sockets; each socket has 15 cores and 30 hardware threads. Running Atlas in "TSP mode" (logging enabled but synchronous flushing disabled) compared with unfortified code reduces throughput by roughly 35% on the desktop and by roughly 30% on the server. This is the price we pay for using Atlas to ensure consistent recovery when TSP is available. When TSP is *not* available Atlas must synchronously flush log entries, and the throughput reduction resulting from Atlas fortification increases to 57% on the desktop and 50% on the server. Comparing the throughput of TSP vs. non-TSP modes of Atlas, we see that TSP increases throughput by 49% on the desktop machine and 42% on the server.

6. CONCLUSIONS

Timely Sufficient Persistence brings substantial benefits when application fault tolerance requirements and available hardware and system software support enable TSP. Our experience with both real applications [3] and small benchmarks (Section 5.2) shows that TSP designs outperform their non-TSP counterparts by wide margins. Remarkably, readily implementable TSP designs for non-blocking algorithms can sometimes completely eliminate runtime overheads while satisfying stringent fault tolerance requirements. Looking forward, we believe that TSP points the way to efficient tradeoffs among runtime overheads, fault tolerance objectives, and hardware and system software support.

7. REFERENCES

[1] G. W. Burr et al. Overview of candidate device technologies for storage-class memory. *IBM J. of Research & Development*, 52(4.5), 2008.

[2] D. R. Chakrabarti and H.-J. Boehm. Durability semantics for lock-based multithreaded programs. In *Hot Topics in Parallelism (HotPar)*, 2013.

[3] D. R. Chakrabarti et al. Atlas: Leveraging locks for non-volatile memory consistency. In *OOPSLA*, 2014.

[4] V. Chidambaram et al. Optimistic crash consistency. In *SOSP*, 2013.

[5] J. Condit et al. Better I/O through byte-addressable, persistent memory. In *SOSP*, 2009.

[6] F. J. Corbató and V. A. Vyssotsky. Introduction and overview of the Multics system. In *Fall Joint Computer Conference, Part I.* ACM, 1965.

[7] J. Dybnis. Non-blocking data structures library for x86 and x86-64, Apr. 2009. `https://code.google.com/p/nbds/`.

[8] K. Fraser and T. Harris. Concurrent programming without locks. *ACM TOCS*, 25(2), May 2007.

[9] HBase. `http://hbase.apache.org`.

[10] G. Heiser et al. Rapilog: reducing system complexity through verification. In *EuroSys*, pages 323–336, 2013.

[11] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, 2008. Pp. 339–349.

[12] D. E. Lowell and P. M. Chen. Free transactions with Rio Vista. In *SOSP*, 1997.

[13] C. Mohan et al. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1), 1992.

[14] D. Narayanan and O. Hodson. Whole-system persistence. In *ASPLOS*, 2012.

[15] F. Nawab et al. Procrastination Beats Prevention. Technical Report HPL-2014-70, HP Labs, 2014.

[16] W. T. Ng and P. M. Chen. The systematic improvement of fault tolerance in the Rio file cache. In *IEEE FTCS*, 1999.

[17] I. Oukid et al. Instant recovery for main-memory databases. In *CIDR*, 2015.

[18] S. Park, T. Kelly, and K. Shen. Failure-atomic `msync()`. In *EuroSys*, 2013.

[19] S. Pelley et al. Do query optimizers need to be SSD-aware? In *ADMS@VLDB*, 2011.

[20] S. Pelley et al. Memory persistency. In *ISCA*, 2014.

[21] S. Pelley et al. Storage management in the NVRAM era. In *VLDB*, 2014.

[22] M. Stonebraker et al. The end of an architectural era:(it's time for a complete rewrite). In *VLDB*, 2007.

[23] The Open Group. *Portable Operating System Interface (POSIX) Base Specifications, Issue 7, IEEE Standard 1003.1.* IEEE, 2008. See line 43041 on page 1310 of the PDF version of the standard for the semantics of writes to shared memory mappings.

[24] S. Tu et al. Speedy transactions in multicore in-memory databases. In *SOSP*. ACM, 2013.

[25] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *ASPLOS*, 2011.

[26] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. *PVLDB*, 7(10), 2014.

[27] S. J. White and D. J. DeWitt. Quickstore: A high performance mapped object store. In *VLDB*, 1995.

[28] J. Zhao et al. Kiln: Closing the performance gap between systems with and without persistence support. In *MICRO*, 2013.